TRiGlobe Part 2 – Connect to webservice

# NSURLConnection

Moving on to Part 2 of TRiGlobe.

Ok now we are going to connect to our webservice and fetch some actual data. First let’s add some properties to our UsersListViewController.m:

@interface UsersListViewController () <NSURLConnectionDelegate>

@property (nonatomic, strong) NSMutableArray \*testArray;

@property (nonatomic, strong) NSMutableData \*buffer;

@property (strong, nonatomic) IBOutlet UIActivityIndicatorView \*spinner;

@property (nonatomic, strong) NSURLConnection \*myConnection;

Now we are going to use NSURLConnection to create a web connection and fetch some data, so let’s change our viewDidLoad method to look like this:

- (void)viewDidLoad{

[super viewDidLoad];

// Animate the spinner

[self.spinner startAnimating];

// Create the URL & URLRequest

NSURL \*myURL = [NSURL URLWithString:@"http://www.yourserver.com/iglobe/getusers.php"];

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

// Create the connection

self.myConnection = [NSURLConnection connectionWithRequest:myRequest delegate:self];

//Test to make sure the connection worked

if (self.myConnection){

self.buffer = [NSMutableData data];

[self.myConnection start];

}else{

NSLog(@"Connection Failed");

}

}

Ok now NSURLConenction has 4 delegate methods you must implement. So first, let’s make sure to add the <NSURLConnectionDelegate> to our @interface line for UsersListViewController.h (or in .m – just make sure it’s the @interface line and not the @implementation line). Now let’s add the first method:

# pragma NSURLConnection Methods

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

[self.buffer setLength:0];

}

This method is called when the app receives a response from the server. We will simply reset the buffer length. Now we must deal with each time the app receives data from the server:

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

[self.buffer appendData:data];

}

We simply append the received data to the existing buffer data. Let’s handle any error response from the server as well:

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

// Do cleanup

self.myConnection = nil;

self.buffer = nil;

// Inform the user, most likely in a UIAlert

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

[error localizedDescription],

[[error userInfo] objectForKey:NSURLErrorFailingURLStringErrorKey]);

}

Finally let’s implement the method that is called when the connection finishes loading data from the web server:

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

NSLog(@"Succeeded!");

//Create a queue and dispatch the parsing of the data

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0), ^{

// Parse the data from JSON to an array

NSError \*error = nil;

NSArray \*jsonString = [NSJSONSerialization JSONObjectWithData:\_buffer options:NSJSONReadingMutableContainers error:&error];

// Return to the main queue to handle the data & UI

dispatch\_async(dispatch\_get\_main\_queue(), ^{

//Check if error or not

if (!error) {

//If no error then PROCESS ARRAY

self.testArray = [[NSMutableArray alloc] initWithCapacity:50];

for (NSDictionary \*tempDictionary in jsonString) { // Extract each dictionary’s username & put it into our array

[self.testArray addObject:[tempDictionary objectForKey:@"username"]];

}

// Call reload in order to refresh the tableview

[self.tableView reloadData];

}else{

NSLog(@"ERROR %@", [error localizedDescription]);

}

//Stop animating the spinner

[self.spinner stopAnimating];

// Do cleanup

self.myConnection = nil;

self.buffer = nil;

});

});

}

Well that’s quite a bit of code but its actually pretty simple. The important bit is that we have seen the web response logged in our console thus we know its an array. We also know the array has dictionaries at each index. Therefore we must loop or iterate through each NSDictionary entry in the array and fetch the “username” key’s value or object. We add that object to our self.testArray with each new iteration. In the end we refresh our tableview to use the newly populated self.testArray with our usernames.

Great so we can read information from our webservice. As we will see later, reading more complex data from the webservice is just a matter of creating a more complex request string on the iOS side and putting it together with some logic in the php server side. This is one way to fetch data, using NSURLConnection directly in a viewDidLoad. It’s better than calling NSURLConnection on the main thread, that’s for sure. But we want to make sure our code is re-usable, particularly the web fetch code since it’s probably code we will want to use again in the future for many projects to come. That’s the reason we created the SantiappsHelper class.

So let’s revisit our roadmap:

1. Create web database
2. Create web service back end
3. Create iOS front end (Storyboard)
4. Fetch Data
   1. **NSURLConnection**
   2. GCD & Completion Blocks
5. Add Bump API
6. Throw in Social

The Users class is simply a container for our individual players. Its interface looks

like this:

#import <Foundation/Foundation.h>

@interface Users : NSObject {

}

@property (nonatomic,copy) NSString \*userName;

@property (nonatomic,copy) NSString \*userPoints;

-(id)initWithUserName:(NSString\*)userName userPoints:(NSString\*)userPoints;

@end

and its implementation looks like this:

#import "Users.h"

@implementation Users

-(id)initWithUserName:(NSString\*)nameOfUser userPoints:(NSString\*)userPoints;

{

if ( (self = [super init]) == nil )

return nil;

self.userName = nameOfUser;

self.userPoints = userPoints;

return self;

}

@end

It’s a very simple implementation much like the Tag Class we will soon see.

# HelperClass – GCD & Completion Blocks

So we want to make our code a bit neater and more portable. We will achieve this by creating our SantiappsHelper Class.

This helper class will concentrate on fetching data from the web. Its very similar to other class files you have worked with before but basically it has only Class Methods.

#import <Foundation/Foundation.h>

#import "Tag.h"

typedef void (^Handler)(NSArray \*users);

typedef void (^Handler2)(NSArray \*points);

typedef void (^Handler3)(NSArray \*usersPointsArray);

@interface SantiappsHelper : NSObject {

}

+(void)fetchUsersWithCompletionHandler:(Handler)handler;

+(void)fetchPointForUsersArray:(NSArray\*)usersArray WithCompletionHandler:(Handler3)handler;

+ (BOOL)postNewTag:(Tag\*)passingObject;// from gamebumpconnector

@end

Ok these 3 methods do the following: (1) fetchUsersWithCompletionHandler will basically fetch the list of users in the game, (2) fetchPointForUsersArray: WithCompletionHandler: will fetch the points for all users and (3) postNewTag will create a new location tag for a particular user.

The idea here is that the user will exchange tokens or tags with another user. Physically the two users at the same location (obviously) will bump phones to initiate a tag worth 2 points. As the data is exchanged, each user account will post a 2-point value tag to the database. Originally the game was built to create tags individually at locations but you cant very well call it a game of Tag if you don’t tag a second user, can you? ☺

Let’s review what these methods do:

// THIS METHOD FETCHES USER ARRAY

+(void)fetchUsersWithCompletionHandler:(Handler)handler {

NSString \*urlString = [NSString stringWithFormat:@"http://www.myserver.com/myApp/getusers.php"];

NSURL \*url = [NSURL URLWithString:urlString];

NSMutableURLRequest \*request = [NSMutableURLRequest requestWithURL:url cachePolicy:NSURLRequestReloadIgnoringLocalAndRemoteCacheData timeoutInterval:10];

[request setHTTPMethod: @"GET"];

\_\_block NSArray \*usersArray = [[NSArray alloc] init];

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND, 0), ^{

// Peform the request

NSURLResponse \*response;

NSError \*error = nil;

NSData \*receivedData = [NSURLConnection sendSynchronousRequest:request

returningResponse:&response

error:&error];

if (error) {

// Deal with your error

if ([response isKindOfClass:[NSHTTPURLResponse class]]) {

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

NSLog(@"HTTP Error: %d %@", httpResponse.statusCode, error);

return;

}

NSLog(@"Error %@", error);

return;

}

NSString \*responseString = [[NSString alloc] initWithData:receivedData encoding:NSUTF8StringEncoding];

//NSLog(@"responseString fetchUsers %@", responseString);

usersArray = [NSJSONSerialization JSONObjectWithData:[responseString dataUsingEncoding:NSASCIIStringEncoding] options:0 error:nil];

//Returns handler

if (handler) {

dispatch\_async(dispatch\_get\_main\_queue(), ^{

handler(usersArray);

});

}

});

}

//Fetches points for users array

+(void)fetchPointForUsersArray:(NSArray\*)usersArray WithCompletionHandler:(Handler3)handler{

NSError \*error = nil;

NSData \*data = [NSJSONSerialization dataWithJSONObject:usersArray options:0 error:&error];

if (error)

NSLog(@"%s: JSON encode error: %@", \_\_FUNCTION\_\_, error);

// create the request

NSURL \*url = [NSURL URLWithString:@"http:/ /www.myserver.com/myApp/readpointsforarray.php"];

NSMutableURLRequest \*request = [NSMutableURLRequest requestWithURL:url];

[request setHTTPMethod:@"POST"];

[request setValue:@"application/json; charset=utf-8" forHTTPHeaderField:@"Content-Type"];

[request setHTTPBody:data];

\_\_block NSArray \*pointsArray = [[NSArray alloc] init];

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND, 0), ^{

// Peform the request

NSURLResponse \*response;

NSError \*error = nil;

// issue the request

NSData \*returnData = [NSURLConnection sendSynchronousRequest:request returningResponse:&response error:&error];

if (error) {

// Deal with your error

if ([response isKindOfClass:[NSHTTPURLResponse class]]) {

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

NSLog(@"HTTP Error: %d %@", httpResponse.statusCode, error);

//return;

}

NSLog(@"Error %@", error);

//return;

}

NSString \*responseString = [[NSString alloc] initWithData:returnData encoding:NSUTF8StringEncoding];

//NSLog(@"asyncrhonous: %@",responseString);

pointsArray = [NSJSONSerialization JSONObjectWithData:[responseString dataUsingEncoding:NSASCIIStringEncoding] options:0 error:nil];

//NSLog(@"pointsArray %@", pointsArray);

if (handler){

dispatch\_async(dispatch\_get\_main\_queue(), ^{

handler(pointsArray);

});

}

});

}

// Called from MKViewController, creates shared tags points=2

+ (BOOL)postNewTag:(Tag\*)passingObject{

//1. Log the tag for verification first

NSLog(@"passingObject:%@,%@,%@,%@,%@",passingObject.sender, passingObject.receiver, passingObject.rglatitude, passingObject.rglongitude, passingObject.rgcountry);

//NSLog(@"tagReceived:%@,%@,%@,%@",tagReceived.originUdid, tagReceived.destintyUdid, tagReceived.rglatitude, tagReceived.rglongitude);

//2.REBUILD status string from passingObject

NSString \*s1 = [[NSString alloc] initWithFormat:@"sender=%@&latitude=%@&longitude=%@&country=%@&receiver=%@&points=2",passingObject.sender,passingObject.rglatitude,passingObject.rglongitude,passingObject.rgcountry,passingObject.receiver];

//3. Post tag to cloud

NSData \*postData = [s1 dataUsingEncoding:NSUTF8StringEncoding allowLossyConversion:YES];

NSString \*postLength = [NSString stringWithFormat:@"%d", [postData length]];

NSMutableURLRequest \*request = [[[NSMutableURLRequest alloc] init] autorelease];

NSURL \*url = [NSURL URLWithString:[NSString stringWithFormat:@"http:/ /www.myserver.com/myApp/writephp.php"]];

[request setURL:url];

[request setHTTPMethod:@"POST"];

[request setValue:postLength forHTTPHeaderField:@"Content-Length"];

[request setValue:@"application/x-www-form-urlencoded" forHTTPHeaderField:@"Content-Type"];

[request setHTTPBody:postData];

NSURLResponse \*response;

NSError \*error;

// We should probably be parsing the data returned by this call, for now just check the error.

[NSURLConnection sendSynchronousRequest:request returningResponse:&response error:&error];

NSLog(@"success!");

[s1 release];

return (error == nil);

}

fetchUsersWithCompletionHandler:

If you noticed at the beginning of our helper class we defined 3 typedefs. This just means we are defining a type, which we called Handler. We will see in a minute what these are used for. So basically this method creates a GET type request, which will be used to GET our users from the webservice we created. It creates a block array to be used inside the block to store our returned data. Before executing the block, which contains the NSURLConnection method call, we dispatch the operation to the background queue and execute it there. This means that a slow or busy server will not block our main thread. The main thread is responsible for drawing operations and user interaction. If a heavy-duty operation such as fetching data from a web server or processing images and video were to be run on the main thread, the user would not be able to interact with the app until that task was complete.

The handler is the typedef that is created once the method has completed. Therefore we check to see if it exists and once it does exist, we call back to the main thread, returning the now populated usersArray. It is that usersArray that will be used to fill our list of users in our tableview.

fetchPointForUsersArray: WithCompletionHandler

This method is basically responsible for sending in an array of users and fetching their points back. It talks to the last php file we created and returns the usersArray, which contains a dictionary with users and their points.

postNewTag

The final method is in charge of simply logging a newly created tag and then posting it to the database. So if you step back a little bit, you can see we have one method for writing data to the database (postNewTag) and two methods for reading data from the database (fetchUsers… and fetchPointForUsersArray…). We read and write points (or Tags) and read users. We must have a way to write users. We will cover this later on in a class called ModalViewController which will accept a login from a user o create a new user. So let’s review where we are:

1. Create web database
2. Create web service back end
3. Create iOS front end (Storyboard)
4. Fetch Data
   1. NSURLConnection
   2. **GCD & Completion Blocks**
5. Connecting
6. Tagging
7. Add Bump API
8. Throw in Social

Before we bump phones and exchange tags we must be able to create tags. So let’s create a Tag Class and its ViewController and plot them on the map view. Once we have that we will then save our user info to our app and be ready to bump!

So in the next part we will look into creating tags and storing them so we can exchange them.