

Building a Basic iOS App

This workshop will teach you the basics about how to build a basic iOS app using Xcode IDE. First, we will go over how to set up Xcode, and creating a Single View application. This will allow you to create both the front-end and back-end of the application in one place, and then run your app on the included simulator or your own iPhone.

We will build an app that saves certain locations from a map. The places saved will be shown in a table, and when you click on them, you will see where they are on the map. No need for Google Maps anymore!

Things we will cover

- Setup Xcode + Create Project
- Create GUI in Storyboard
- Create Map + Table View Controllers
- Show user's location
- Show map pin on long press
- Show saved locations as map pins
- Add user's location to Table view

Setup

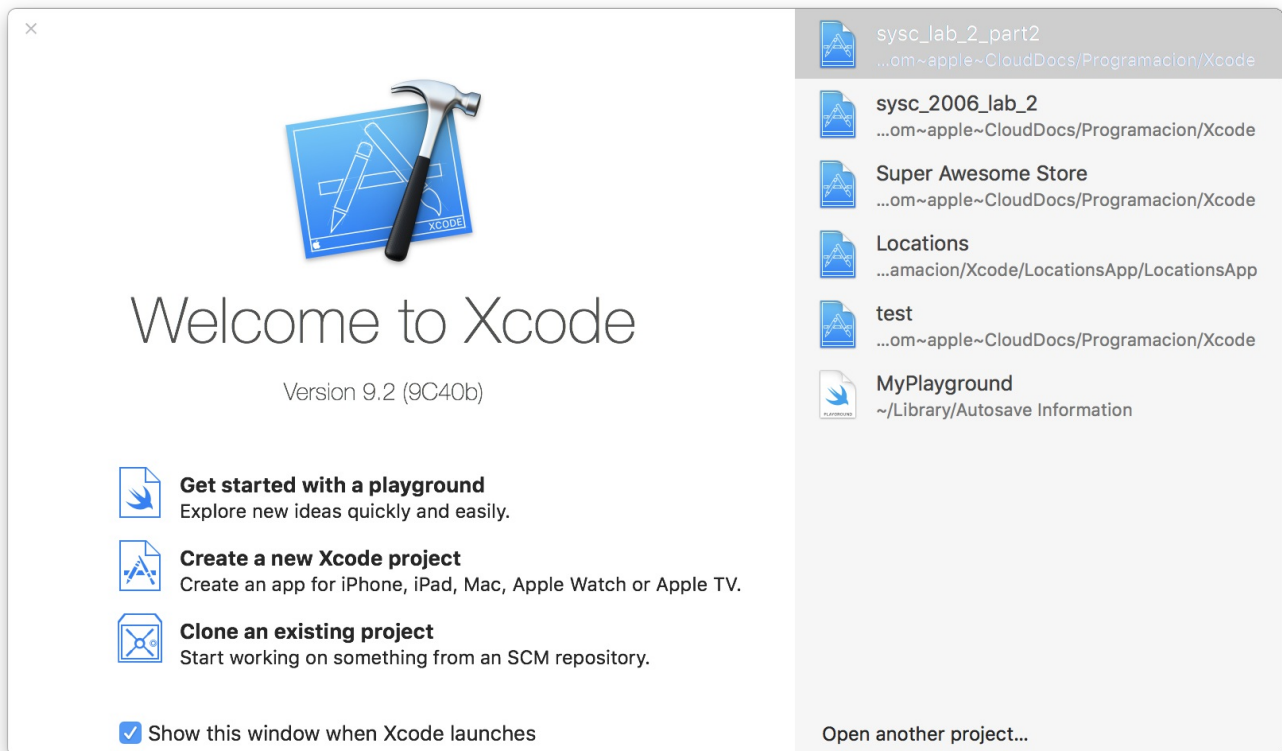
This section covers installing Xcode and creating a single view application project.

Installing Xcode

In order to install Xcode, you must first download it from the [App Store](#). It's a large 5.5 GB file so start the download and go grab a cup of coffee. Once it's done downloading you can the app; you will get a license agreement, press accept.

Creating Xcode Project

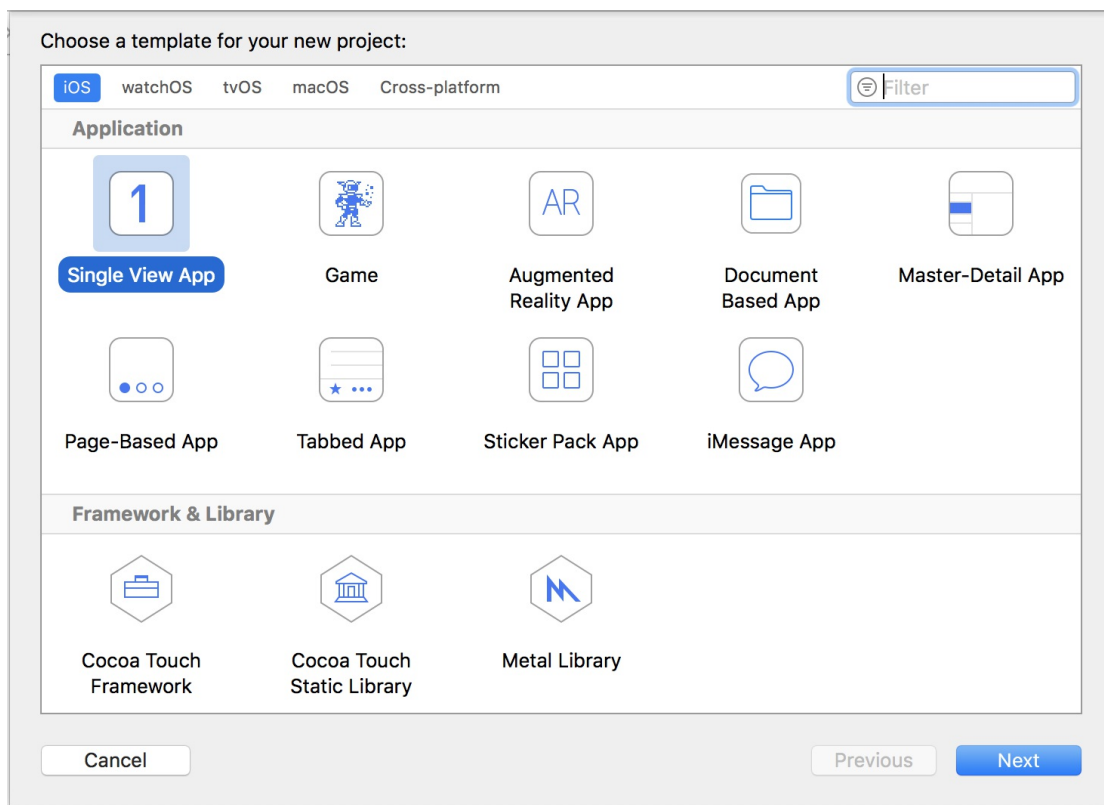
Once you start Xcode you should get a welcome screen. If this window does not appear, or you closed it accidentally, you can press **command + shift + 1**. Also, you can open it through the menubar option: **Window > Welcome to Xcode**.



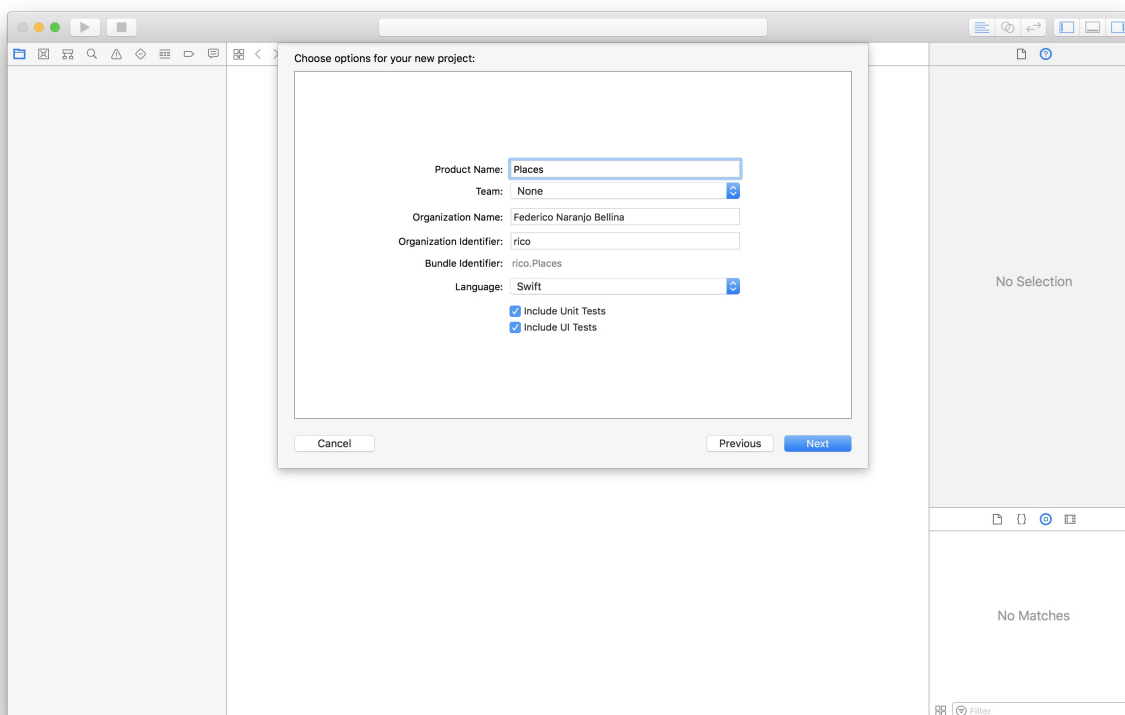
Click on the button to create a new Xcode project, shown below. This project will contain all the files you need to build and run your iOS app.

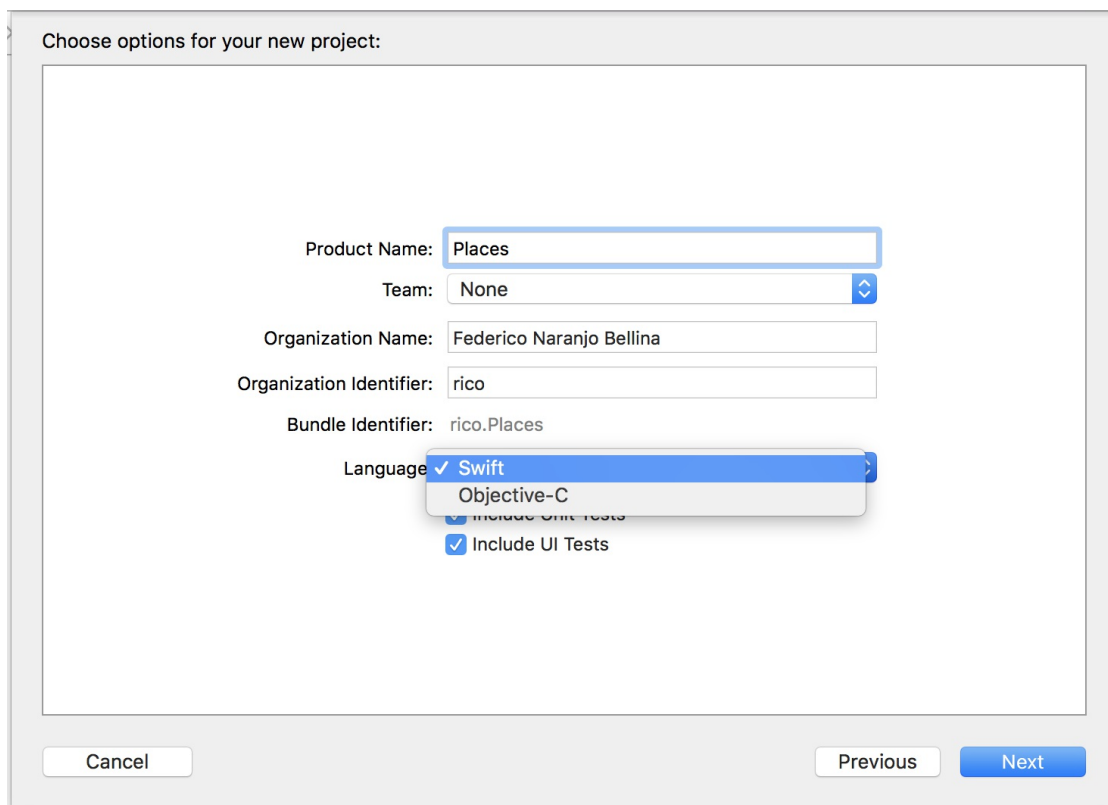


The next screen should give you the options of which type of project you wish to create. We want to create a console line application so select the **Command Line Tool**.

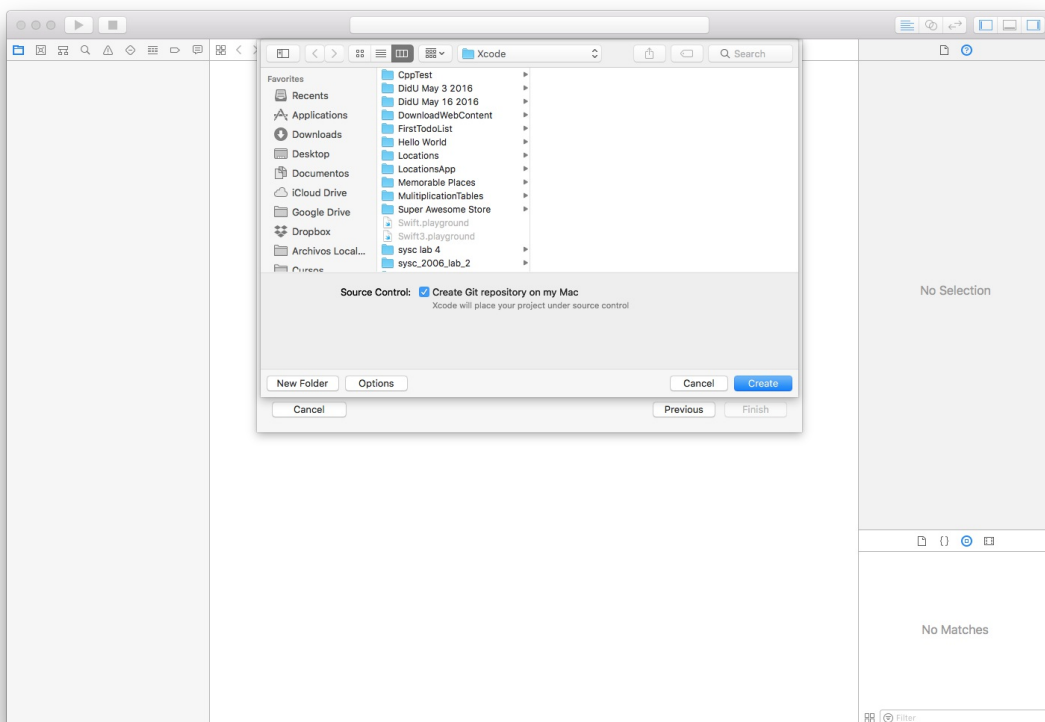


In the next screen you can give your project a name and select the programming language you will be using. Make sure you select the correct one, because there is no way to change it after the project is created. With Xcode you have the option of using **Swift** or **Objective-C**. For our purposes I will use **Swift**. Press **Next**.



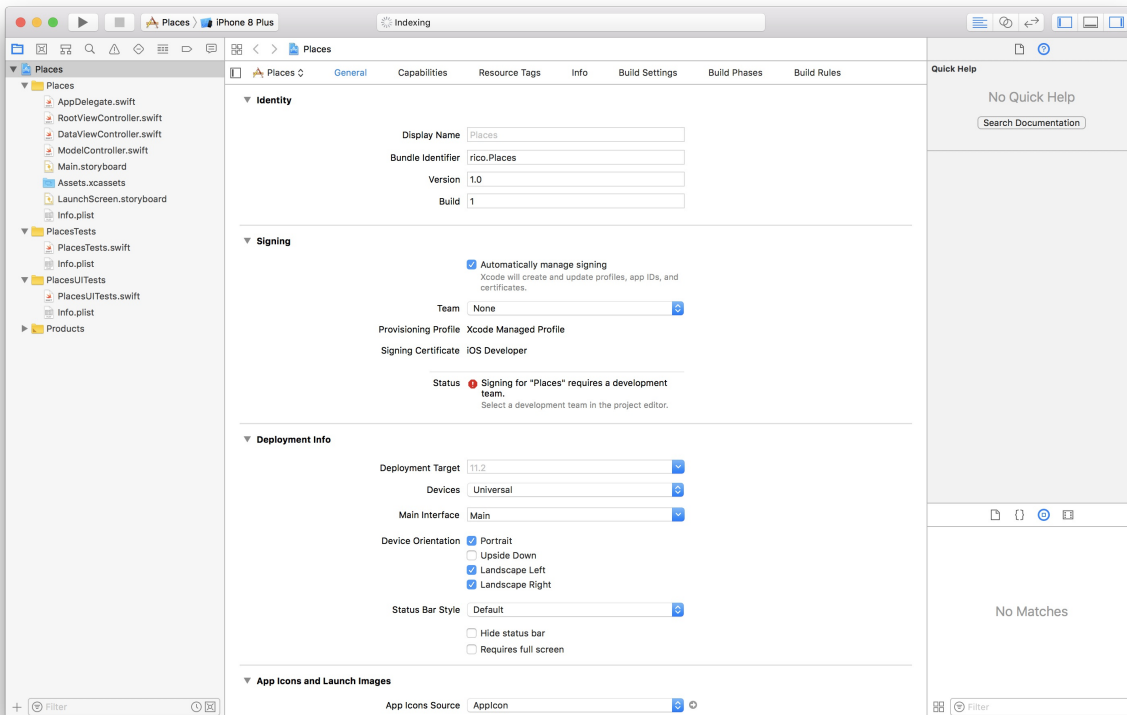


A window will appear asking where you want to situate the files; you can enable source control with Git, useful if you are also uploading to GitHub. Find a location where you want to keep your project and press **Create**.



Introduction to Xcode

Your new project will now open in the *workspace window*. There is a lot of information on the screen, but you can ignore most of it for now. On the left side of Xcode, you will find the *navigator area*. In this pane you can see all of your files that belong to this project.



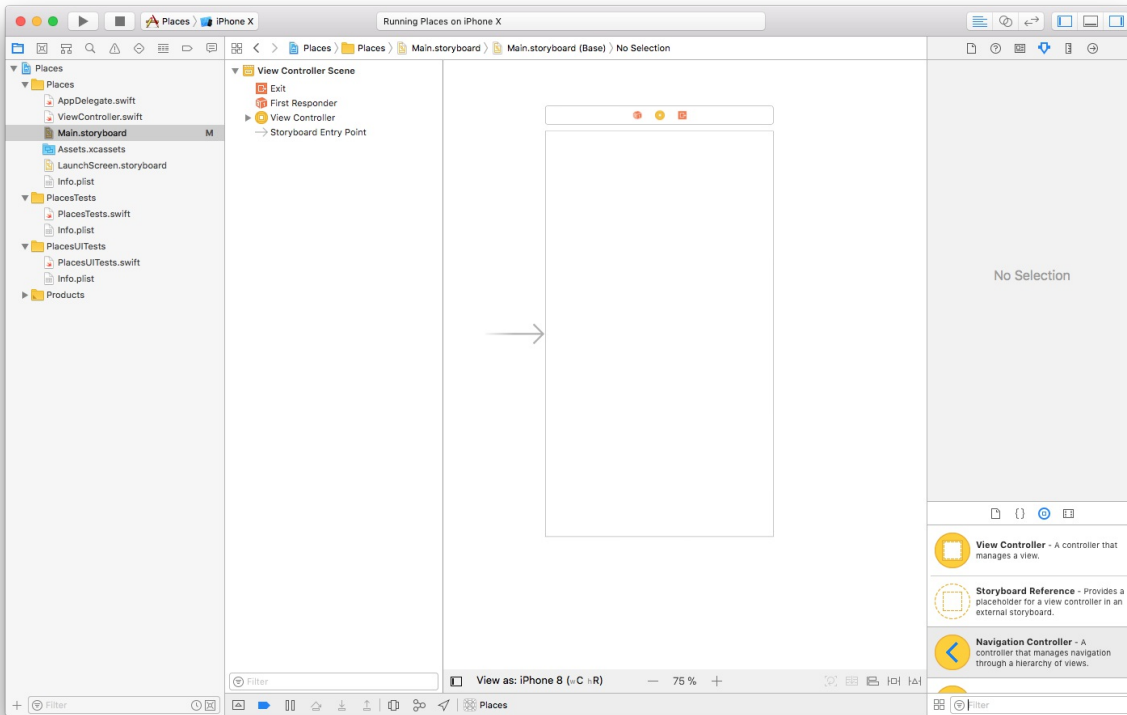
If you want more information on how to use Xcode, you can find a wonderful tutorial by Apple [here](#).

Reading the Storyboard

Select the file named **Main.storyboard**. This is the **Storyboard** for your new app.

The **Storyboard** allows you to create drag and drop UI elements so that you can design the look of your app visually — without having to code the location of every button and text field.

If you want to add a button, all you have to do is drag and drop; if you want to change the colours, you can select it from a colour wheel. This allows you to see what your app will look like without having to build and run everytime you make a change.




Navigation Controller

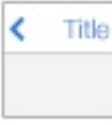
View Controllers are what define what the user will see, and the behaviour of on-screen elements. The Navigation Controller manages a stack of view controllers, providing a drill-down interface. It is used in applications like the **Settings** app, where pressing an option in a Table takes you to another view, with a handy back button in the upper-left corner.

If you want more information about Navigation Controllers, you should look at Apple's official documentation on them [here](#).


We will start by going to the bottom right search field and typing 'navigation'. The first result should be the **Navigation Controller**. Now drag and drop this into the Storyboard workspace. It should look like two View Controllers next to each other.




Navigation Controller - A controller that manages navigation through a hierarchy of views.

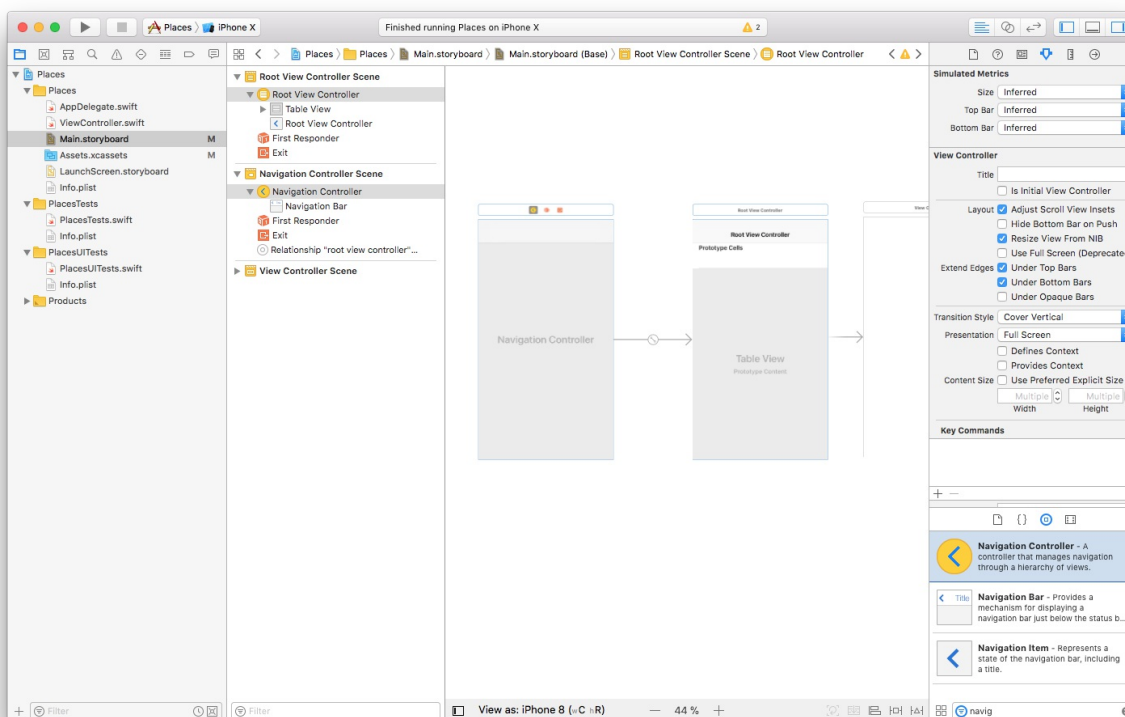


Navigation Bar - Provides a mechanism for displaying a navigation bar just below the status bar.



Navigation Item - Represents a state of the navigation bar, including a title.





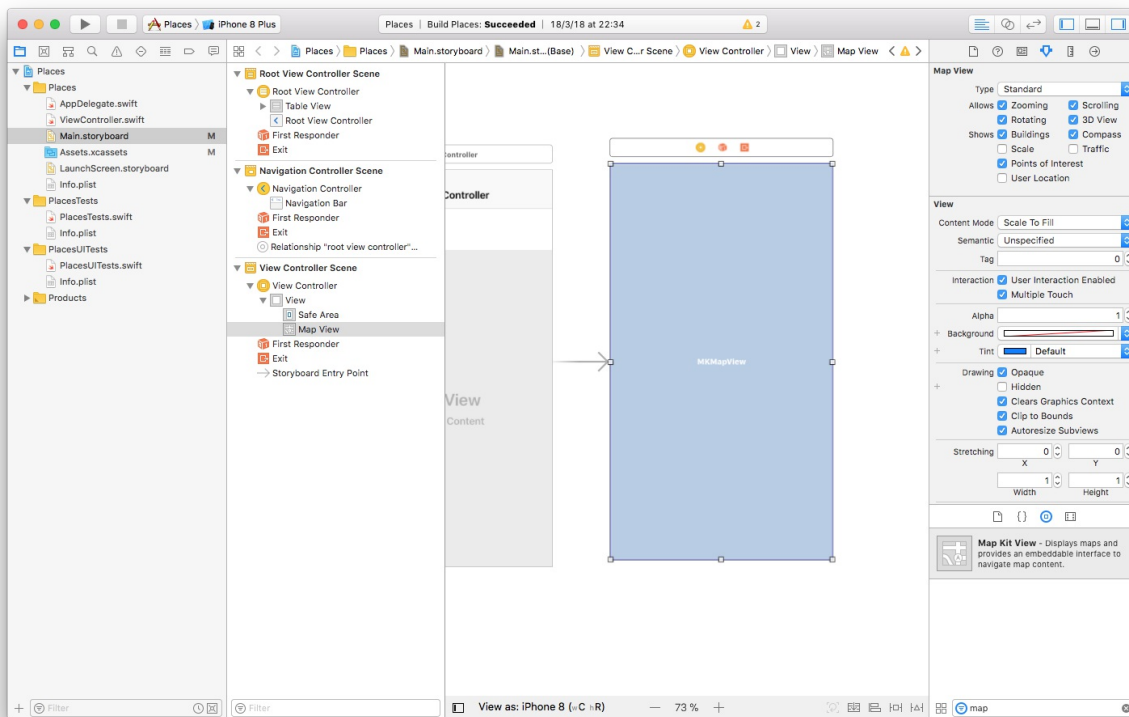
Now there are two view controllers in the Storyboard canvas: the Root View Controller — which was added with the Navigation Controller — and the View Controller which was already in there. The Root View Controller comes with a **Table View**, so this will be where all our stored places are displayed. The other View Controller will become our Map View.

A Table view is simply a view with a series of cells which can be populated with any data. Examples of apps that use this are Twitter, Facebook, or the Mail app.

Map View

The **Map View** allows for quick and easy integration of Apple Maps into your app. It is as easy as simply drag-and-drop onto a View Controller; if you want to show the user's location it only requires a few lines of code.

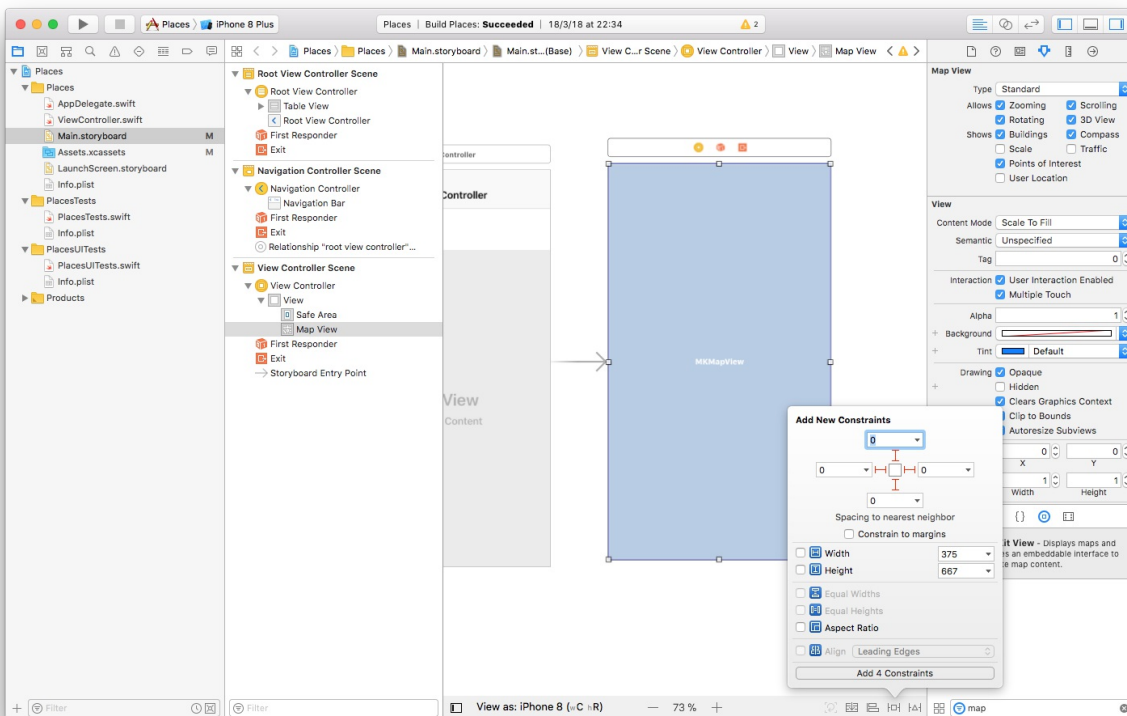
Add a **Map View Kit** to the original View Controller. Change its size to cover the entire View Controller.



Constraints

Constraints are the rules that allow for apps to work on different display sizes. This ensures that users have good experiences on apps whether they have an iPhone SE, iPhone 7, or iPhone X, without developers having to create custom layouts for each screen size. For example, you can make the map view stretch to fill the entire screen or just half the screen, without hardcoding any values based on model of phone.

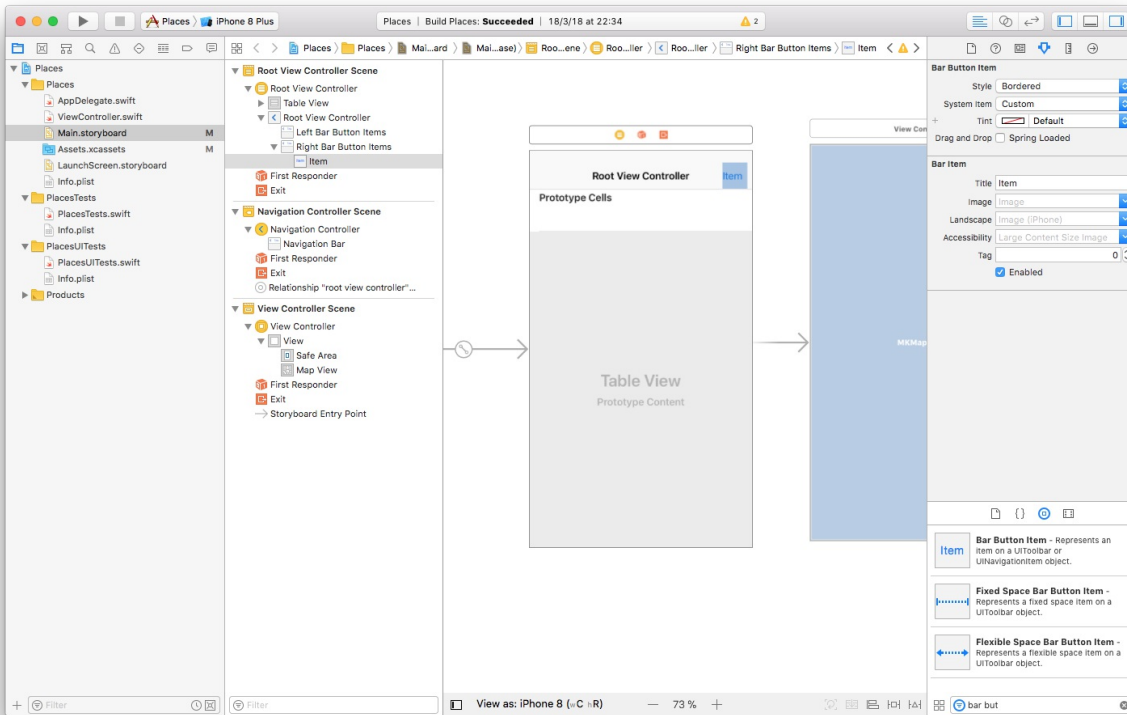
To add constraints to our Map View, press the **Add Constraints** button in the bottom right of the centre panel.



Bar Buttons

Bar buttons are special buttons that you can add to the the navigation bar on the top of a view. For example, when you are in the Photos app, you can add an Album in the bar button in the top-left.

Add a **Bar Button Item** to the top-right of the **Root View Controller**. It should be a button with a label of 'Item'.



Next we're gonna make it an add (+) sign. Go to the **Attributes Inspector** in the right pane.



Change the **System item** value to *Add*. Now the button label should simply be '+'. This button will be used when we want to add a new location to our app.

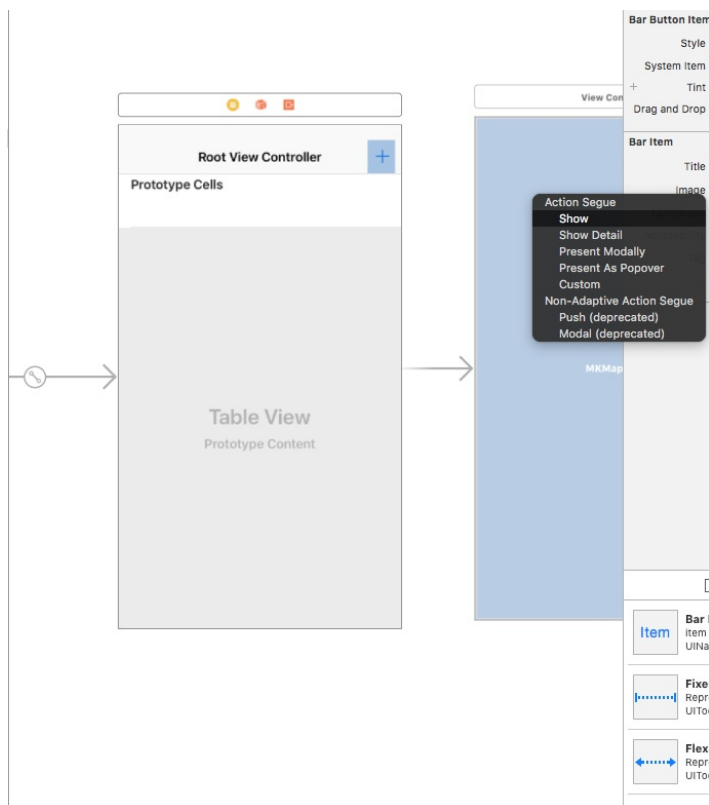
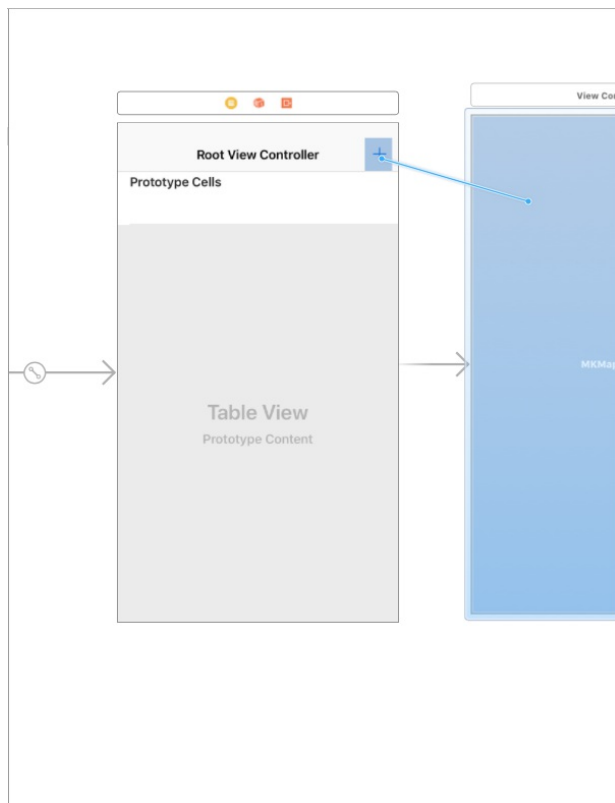
Segues

In order to move from one view to another — say from the main Settings page to General Settings – you need a segue between the views.

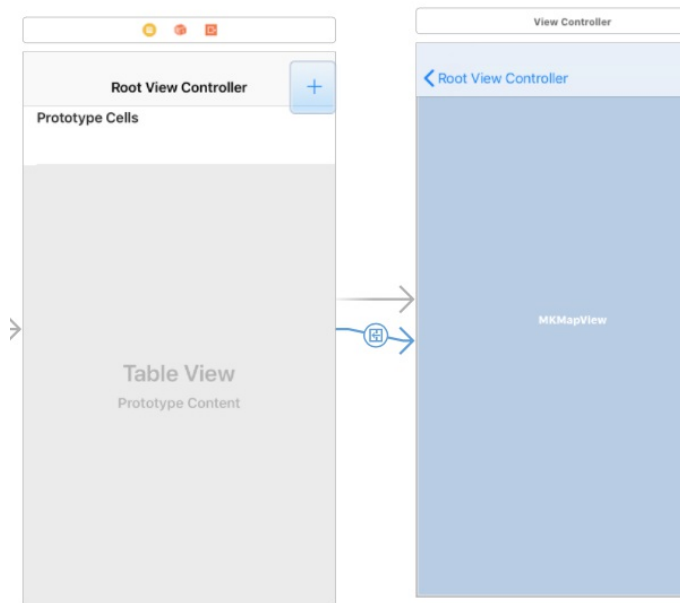
We are going to add a **Segue** from the Table View to the Map View when we press the Add button. This will allow us to add new places to our table.

To add the segue **ctrl + click** on the add button and drag to the Map

View, you should see a blue line extend from the button.



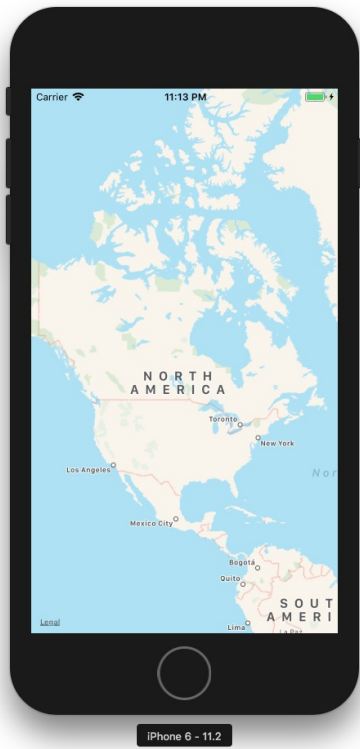
It should add a new arrow, from the Table View to the Map View, with a circle icon.



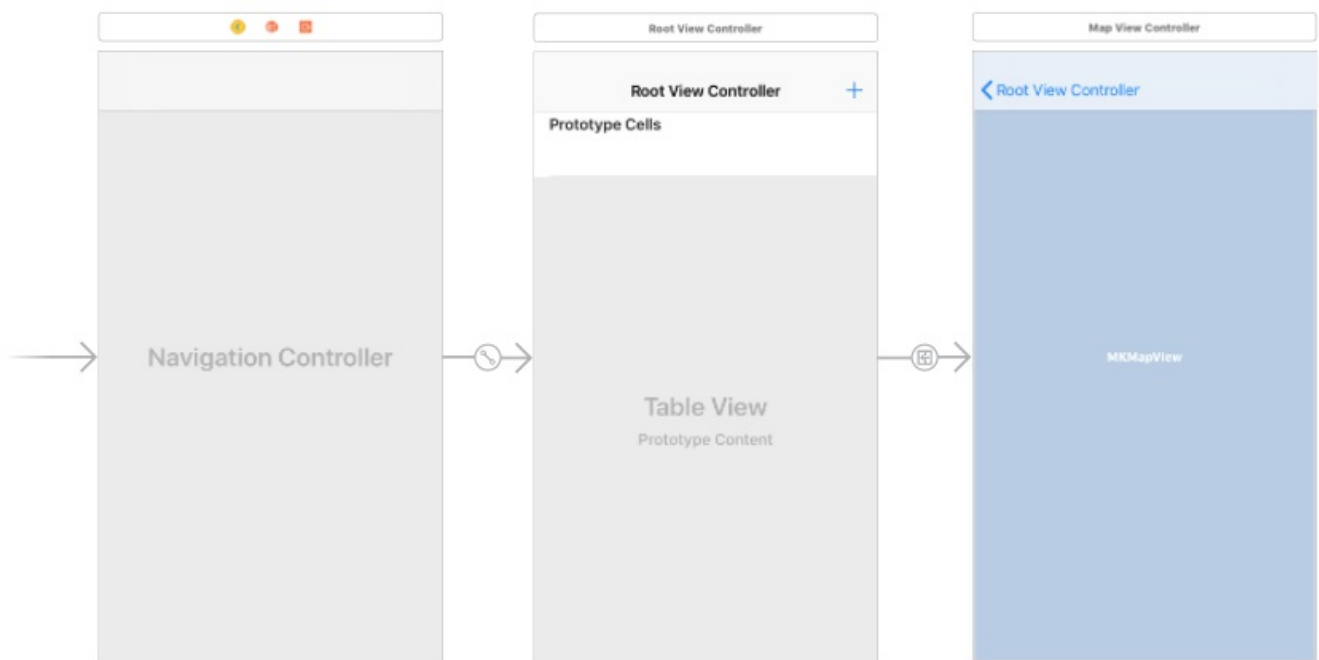
Storyboard Entry Point

The **Storyboard Entry Point** defines which scene or view appears first in the 'story', in this case on app launch. You can see it as the arrow [without a symbol] next to the segue arrow.

Try pressing command + r to run your app.



In order to change the entry point, you can simply drag-and-drop the arrow to the Navigation Controller.



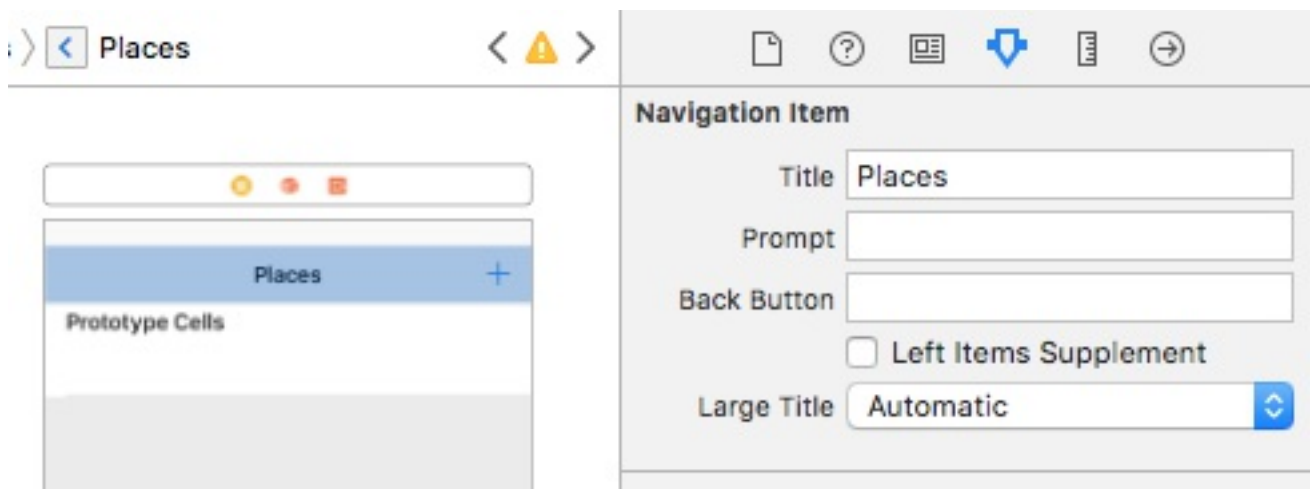
Now our app should open in the Table View. Try running the app again to confirm this. Try pressing the Add button and see what happens.

Modifying the Default Names

Now that we have a basic working app, I think it's time to move on from the generic, un-descriptive, default names.

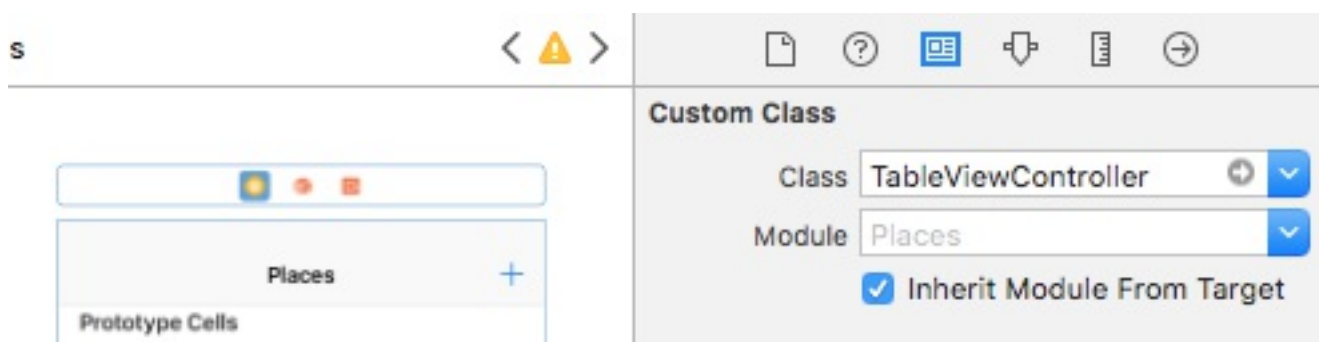
First let's change the name that is displayed in the Table View title bar, since this is the first thing the user will see when they launch the app.

Click on the Root View Controller title [the navigation bar, which has the + button], and change the text to your app name. You can change the content of the **Navigation Item** [the app title bar] in the right panel after selecting it, in the Attributes Inspector.

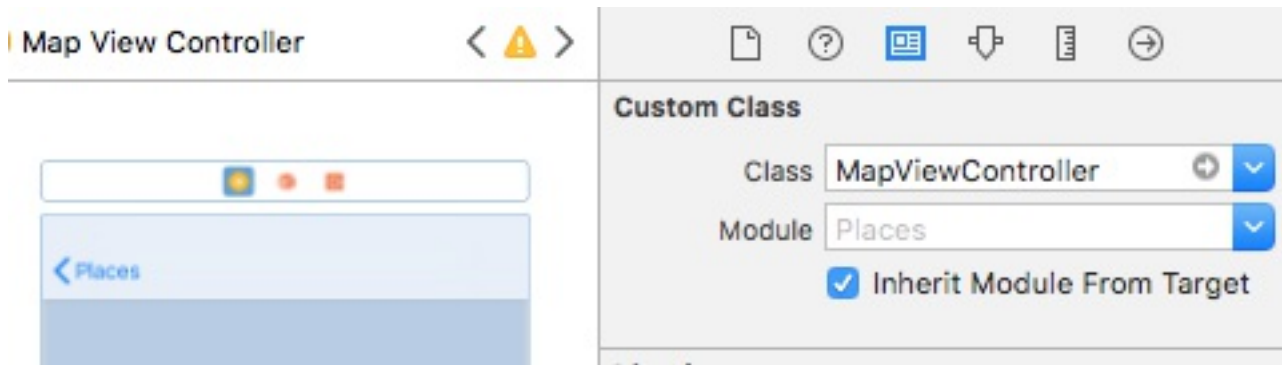


Next we want to change the classes of our View Controllers. We want to do this so we can define the logic for each of these views in the next section.

Select the bar on top of the Table View scene, and go to the identity inspector, and add a Custom Class name. It's best to make it something descriptive, so I went with UITableViewController.



Now do the same with the View Controller.

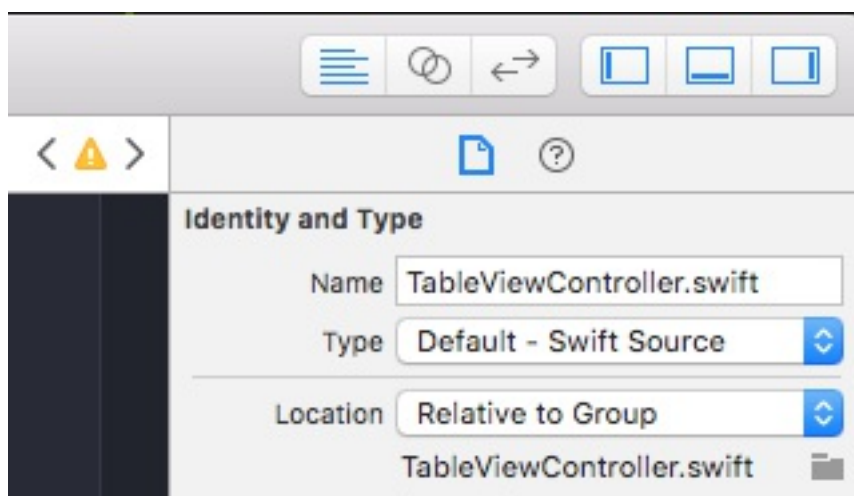


View Controller Files

So we have now defined the UI of our app, but we have yet to create all the logic. We will add this logic in the View Controller swift files.

Xcode already added ViewController.swift file for us; we can use this one for the Map View. We need to create a new one for the Table View.

First go to ViewController.swift and change it's name to TableViewController.swift. You can do this by first selecting the file, and entering the new name in the File inspector on the right side.



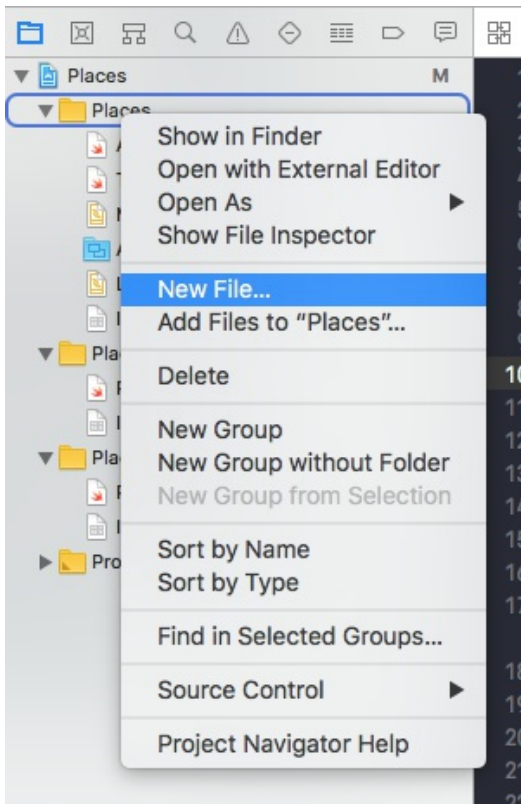
Now rename the **ViewController** Class to **TableViewCellController**. Change the parent class to **UITableViewController** also.

```
class TableViewController: UITableViewController {
```

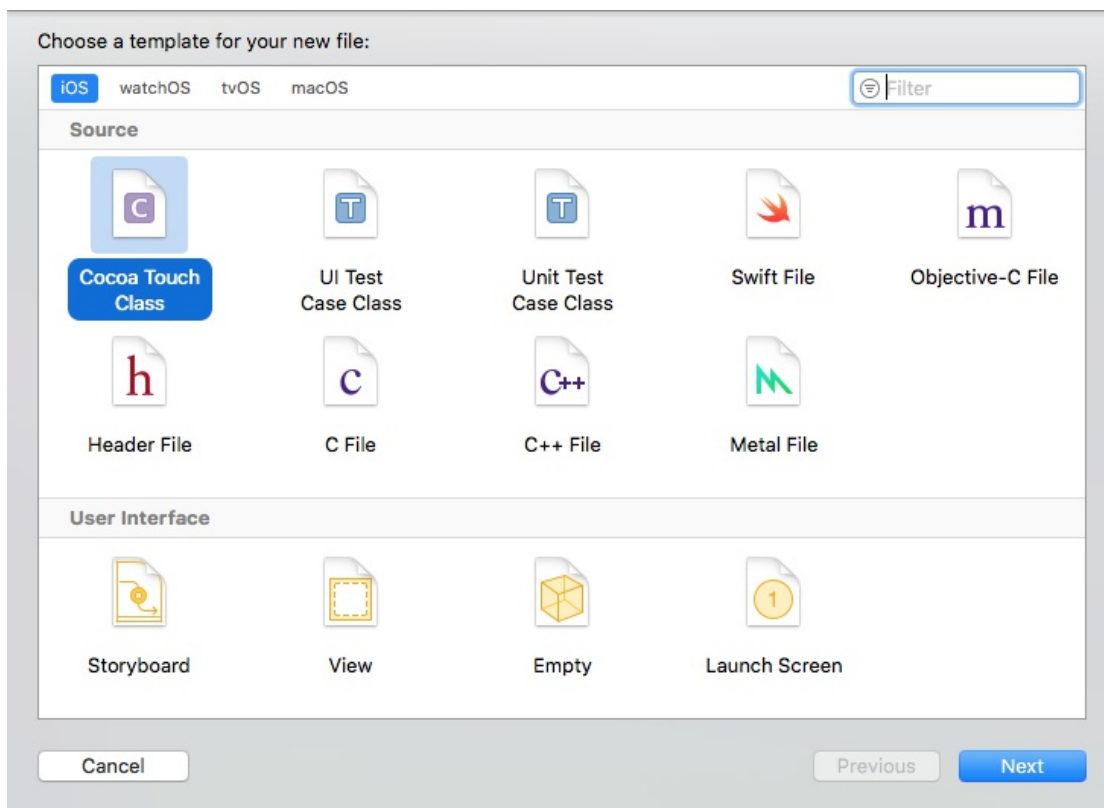
Now let's create a new View Controller file for our Map View.

Add New View Controller File

To add a new View Controller file, right click on the app folder, and select *New File*.

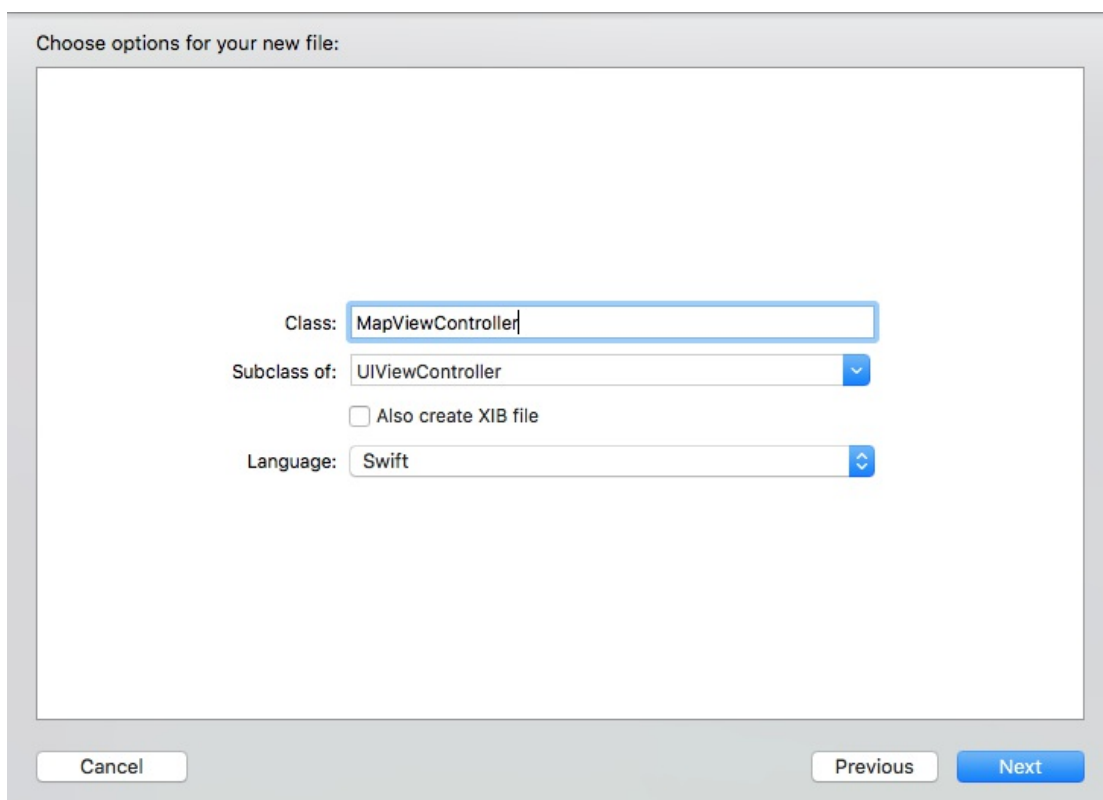


You will get a window asking you what type of file you want to create; we want to create a Cocoa Touch Class file.



The next screen is for specifying the class name, and what subclass it is of. By default it will say NSObject, which is the topmost class [same as the Object class in Java].

Name your new class MapViewController, subclass of UIViewController. You can change the language to Objective-C, but we will stick with Swift.



Press **Next**. A window will appear, to specify where to add this new file. The default location should be fine. Press **Create**.

Add Button Outlets

Now that we have files to store the logic for our view controllers, we need to add logic to them.

First let's add button references [Xcode calls these Outlets], so that we can use the buttons to trigger things, like adding user locations. To do this, we have to go back to the Storyboard.

[sidenote: I added a clear button in the table view to delete all the locations]

Adding button references to the View Controller files is as easy as simply dragging the button to where you want the reference.

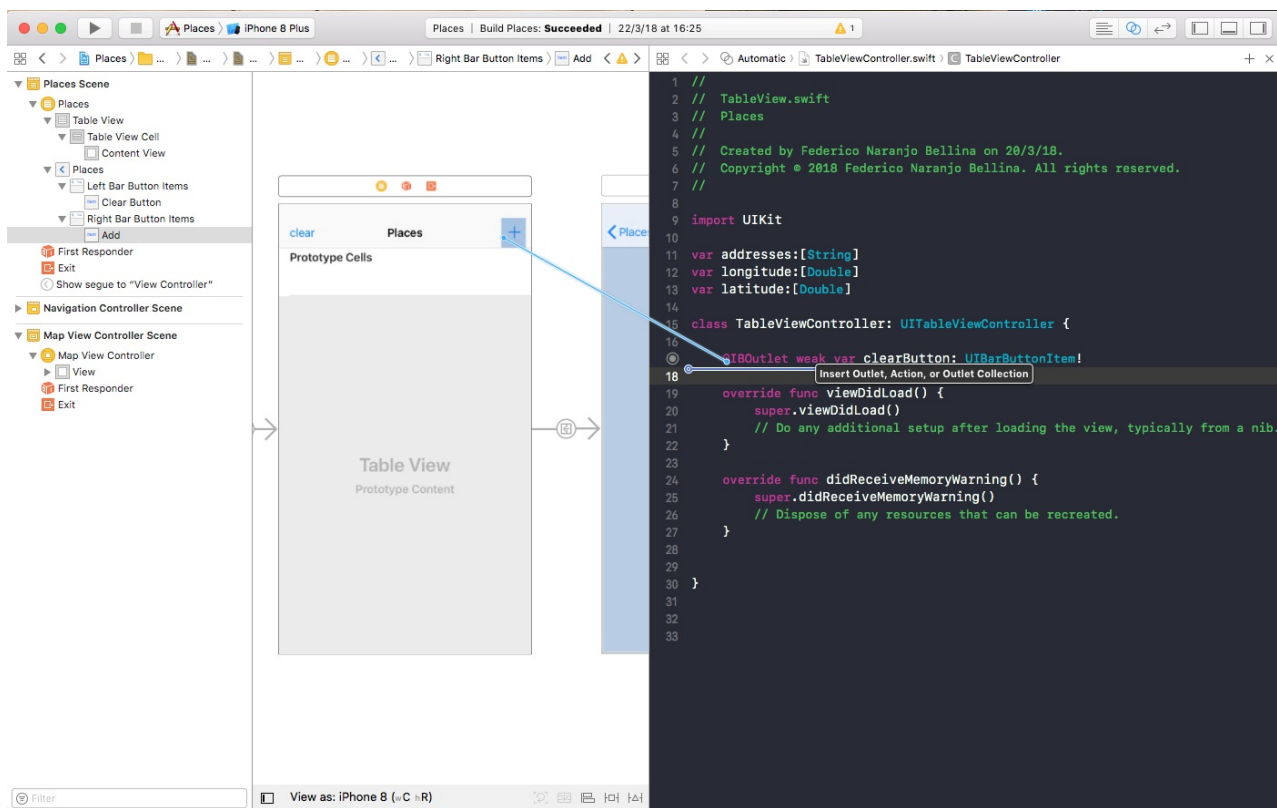
Press the **Assistant Editor** button in the top right.



This will open up the View Controller logic for each selected scene. You might find it easier to hide the Navigator, Debug, and Utilities areas; you can use the buttons on the top right to do that, or you can use the shortcuts:

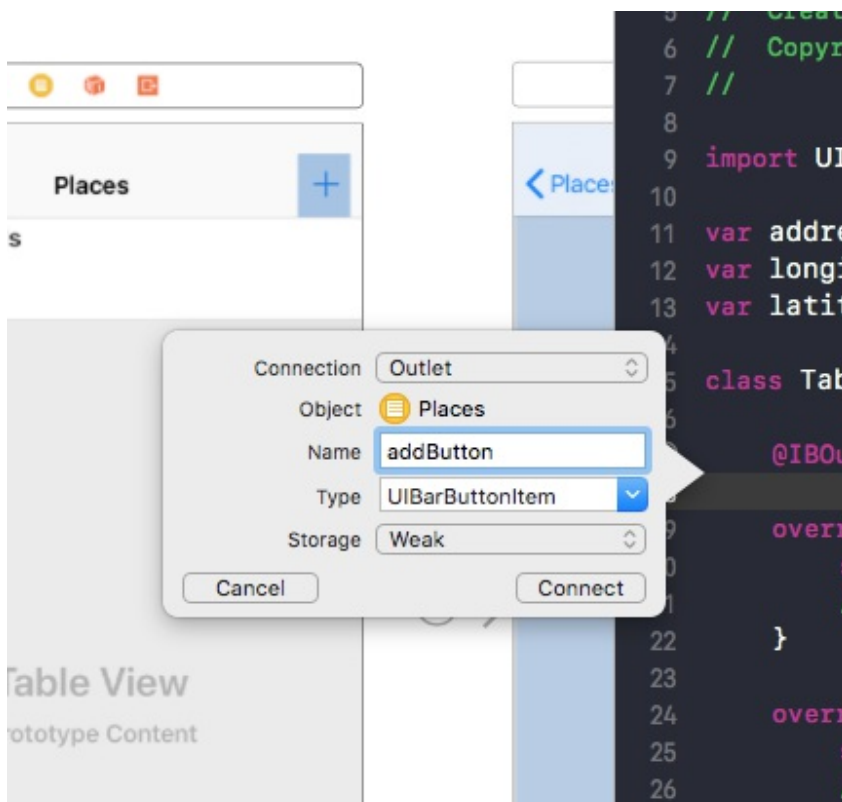
- Project Navigator: command + 0
- Debug: command + shift + y
- Utilities: command + option + 0

To add the Add button to our view controller, hold ctrl and drag the button over to the editor pane.



Place the reference somewhere that is not within a method.

You will be given the option to name your reference and change other characteristics about it. We want to leave it at the default values. Press **Connect**.

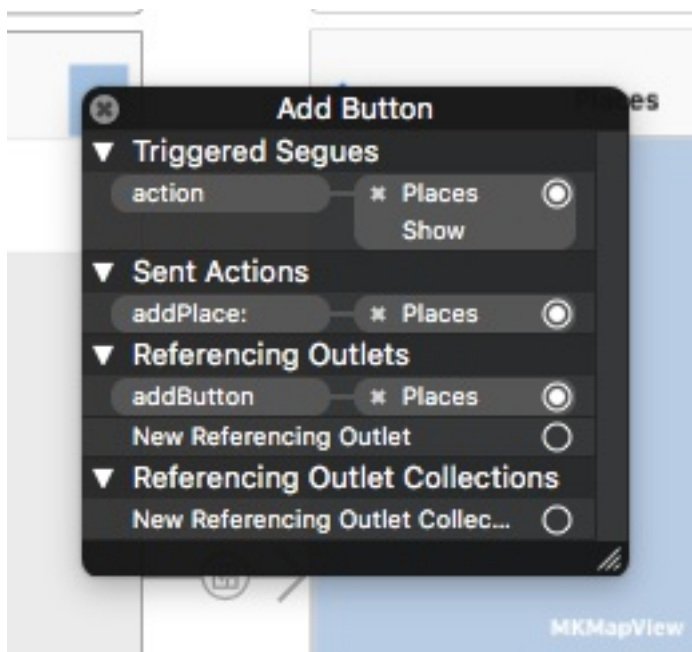


Add an outlet for the MapView as well. This will be important later on.

Now, moving on to the Map View Controller, let's add a button so we can add the current location on screen while in the Map View. To do this, add a **Navigation Item** to the Map View. Once you do this, you can add a **Navigation Bar Item**.

Now add the button outlet for the Add button the same way as before.

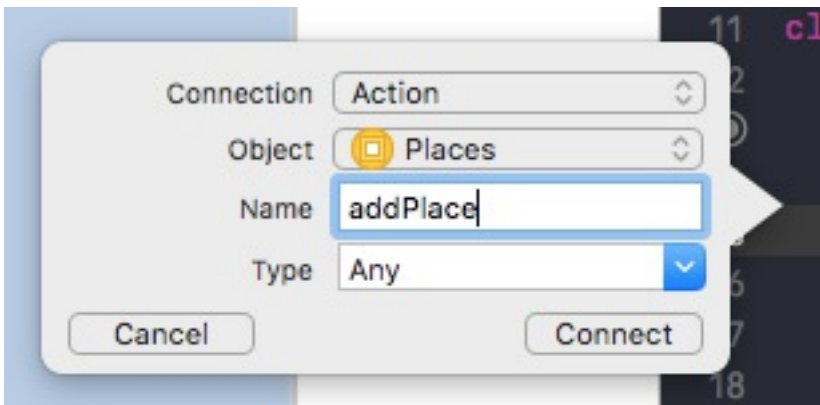
Be careful: if you delete the code for an outlet or action without also removing it from the Storyboard, you will get a Sigthread error, which can be very hard to track down. Make sure to not delete the outlet or action for a button in the swift files, unless you also delete it from Storyboard.



You can delete it by right clicking the button and pressing the 'x' next to the outlet/action name.

Button Actions

In order to define logic to be executed when a button is pressed, we need to define an action. This is done in a very similar way to how we added the button reference. Instead, this time when giving the reference a name, also change the Connection type to from Outlet to **Action**. This will create a method that is triggered when the button is pressed.

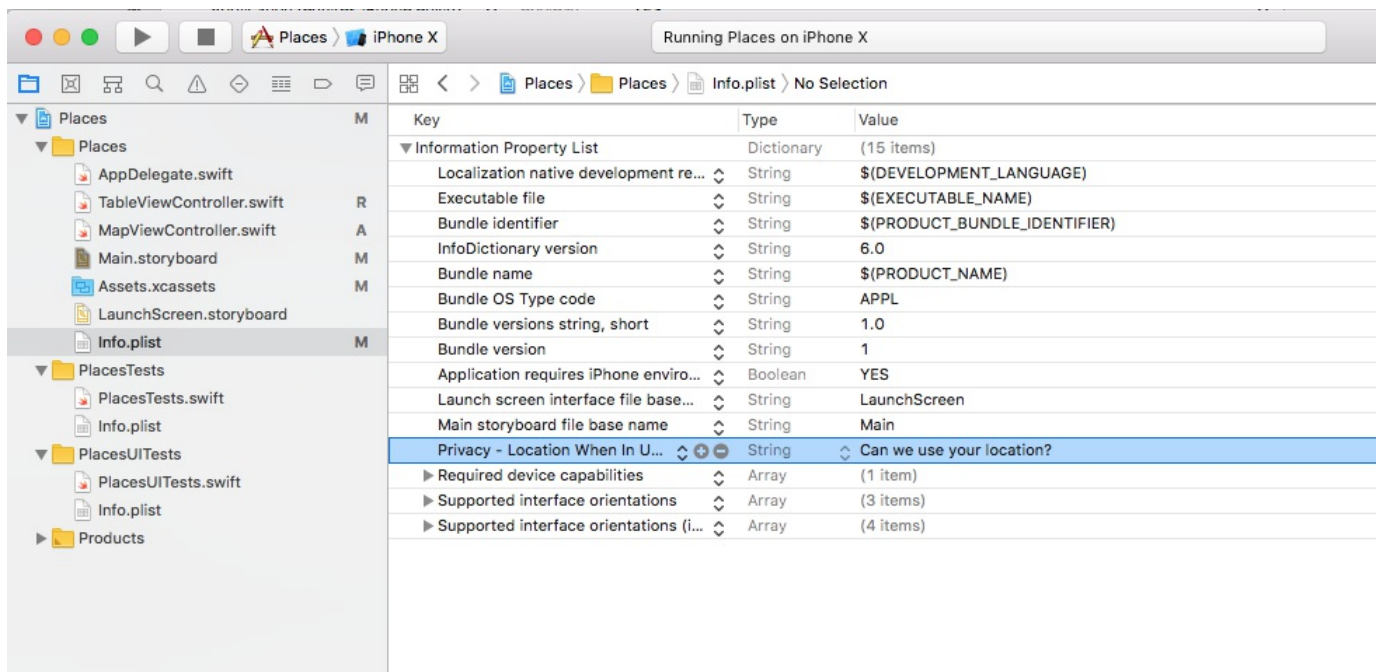


How to show user location in Map View

In order to get the user location, we need to first ask the user's permission. This is done by Apple to ensure no app developers can get access to user data without consent.

Asking Permission

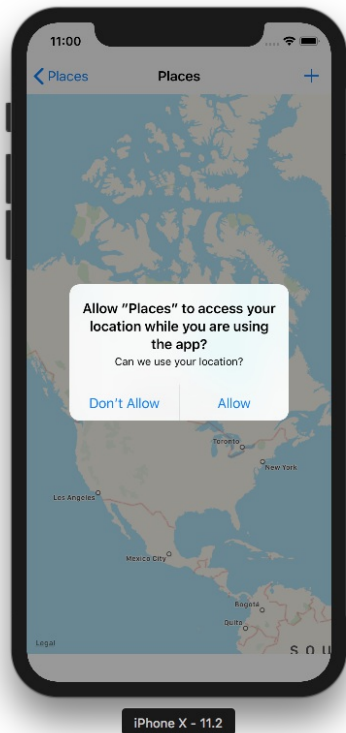
To get a prompt to use the user's GPS location, we need to add a row to the info.plist. You can find this file on the **Project Navigator** on the left hand side. If it is collapsed, you can expand it with command + 0.



Right click and select *Add Row*. Start typing the text below, it should autocomplete it though.

Privacy – Location When In Use Usage Description

Add some text in the Value column. This is the message the user will see when they first go to the Map View. Add a simple message like "We need your location to show your position on the map".



Try running your app and see if you get the pop-up when you go to the Map View.

Showing the blue dot

To show a blue dot showing where the user is, we need to add a few lines of code to the **MapViewController** Class.

First add the following line to the top of the file,

```
import UIKit    // this one should already be there
import MapKit    // add this one
```

Make the class inherit from the **CLLocationManagerDelegate**, and add a **CLLocationManager**. This will manage the coordinates from the phone's sensors.

```
class MapViewController: UIViewController,
CLLocationManagerDelegate {

    let locationManager = CLLocationManager()
```

Now we need to set up the location manager when the view loads.

Go to the viewDidLoad method and add the following lines:

```
/// this runs everytime the view is loaded
override func viewDidLoad() {
    super.viewDidLoad()

    // sets up the delegate
    locationManager.delegate = self
    locationManager.desiredAccuracy =
kCLLocationAccuracyBest
    locationManager.requestWhenInUseAuthorization()
    locationManager.startUpdatingLocation()

    // show user as blue dot
    mapView.showsUserLocation = true
}
```

Now try running your app.

Add the following function,

```
/// Automatically updates the GPS location of user and
where map is centred.
/// This function is called by the system
func locationManager(_ manager:CLLocationManager,
didUpdateLocations locations:[CLLocation]) {
    userLocation = locations[0] // the last user
location
```

```

        // field of view, how zoomed in the map is
        let mapSpan:MKCoordinateSpan =
MKCoordinateSpanMake(0.02,0.02)

        // centre map on the user
        let region =
MKCoordinateRegionMake(userLocation.coordinate,mapSpan)
        self.mapView.setRegion(region, animated: true)
// animated will make make zoom in on location
}

```

Adding User's Location to Table View

To add the user's location to the table view, we need to take the GPS position of the user, use that to get their address, and then we can store that along with their longitude and latitude.

We can use the ReverseGeoCoder method to do this. The reverse geocoder, takes a gps coordinate and gets the address for that location.

Add these two functions to the Map View Controller. The first function takes the a location, retrieves all the information, and creates a map pin.

```

/// takes the CLLocation, and finds the placemark from
reverse geocoder
func getPlacemark(_ cllocation:CLLocation) {
    CLGeocoder().reverseGeocodeLocation(cllocation,
completionHandler: { (placemarks, error ) in

        if error != nil || placemarks == nil ||
placemarks!.count == 0 {
            print(error ?? "Unknown error in geocoder")
            return
        }

        let place = placemarks![0] as CLPlacemark

```



```

        // parse address and coordinates
        let address = self.parseAddress(place)
        let latitude = cllocation.coordinate.latitude
        let longitude = cllocation.coordinate.longitude

        // create map annotation
        let annotation = MKPointAnnotation()
        annotation.coordinate =
CLLocationCoordinate2D(latitude: latitude, longitude:
longitude)
        annotation.title = address
        self.mapView.addAnnotation(annotation)
    })
}

```

This function takes a CLLocation from the location manager from before, and then finds a placemark for this location. The placemark stores all the information like the street name, province, country, and various other data.

Since this function is asynchronous, any function calls inside it need to use the qualifier *self*. By making this function call asynchronous, the app won't appear slow when adding a user location since the app will continue to be responsive while it retrieves the location data.

To parse this information I created a parseAddress function which simply finds the address number [subthoroughfare], street name [thoroughfare], and city [locality] and makes a simple string out of them.

```

/// Parses location from CLPlacemark place and stores
it in address
func parseAddress(_ place: CLPlacemark) -> String {
    var address = ""
    var locality = place.locality ?? ""
    var thoroughfare = place.thoroughfare ?? ""
    var subThoroughfare = place.subThoroughfare ?? ""

```

```

        // if all paramters are empty, only add country
        if subThoroughfare == "" && thoroughfare == "" &&
locality == "" {
            address = place.country ?? "No country or
address found"
            return address
        }

        // add space after section if not empty
        if locality != "" { locality += " " }
        if thoroughfare != "" { thoroughfare += " " }
        if subThoroughfare != "" { subThoroughfare += " "
    }

    address = "\(subThoroughfare)\(thoroughfare)\
(locality)"
    print(address)
    return address
}

```

Try running the app and pressing the '+' button

Adding Map Pins on Long Press

In order to add map pins when you long press on the screen, we need to set up a listener first, called a **UILongPressGestureRecognizer**. Add these lines of code to set it up in the **viewDidLoad** function.

```

// allow long press to add map pin
let uilpgr = UILongPressGestureRecognizer(target: self,
action: #selector(MapVC.addLongPressLocation(_:)))
mapView.addGestureRecognizer(uilpgr)
uilpgr.minimumPressDuration = 0.35

```

We also need to add an action when a long press is held:

```

/// add location where long press
@IBAction func addLongPressLocation(_ sender:
UILongPressGestureRecognizer) {

    // gets location of long press relative to map
    if (sender.state == UIGestureRecognizerState.began)
    {

        let touchPoint = sender.location(in: mapView)
        let newCoordinate = mapView.convert(touchPoint,
toCoordinateFrom: mapView)

        let pressed = CLLocation(latitude:
newCoordinate.latitude, longitude:
newCoordinate.longitude)

        getPlacemark(pressed)
    }
}

```

Try running your app, and holding a long press. See if it creates a map pin of your location.

Showing Places in Table View

Now that we are able to add locations to the map, we want to store these places to the Table View. First we need to store the address and coordinates in global arrays.

Global variables should be ignored but for this workshop we will use them for the sake of time. You should use CoreData instead – perhaps next workshop.

Add these three arrays outside one of the classes in the swift files. I placed mine above the Table View Controller class.

```

var addresses:[String] = [String]()

```

```
var longitudes:[Double] = [Double]()  
var latitudes:[Double] = [Double]()
```

Storing values in Map View

To add the values to these arrays, we need to store the values within the **reverse geocoder method**. This will ensure that they are saved after the values are retrieved and parsed.

I added this line below the creation of the map annotation, but before the `})`.

```
self.savePlace(address: address, longitude: longitude,  
latitude: latitude)
```

Now let's add the function definition:

```
/// store place in arrays  
func savePlace(address:String,  
longitude:CLLocationDegrees,  
latitude:CLLocationDegrees) {  
    addresses.append(address)  
    longitudes.append(longitude)  
    latitudes.append(latitude)  
}
```

This function will add each place to the end of the array.

If we want our map pins to show all of these locations, we can use the global arrays we just created. We can iterate through all the values in the arrays and add each coordinate with the address as the description.

we can just reuse the logic in the reverse geocoder for the creation of the new map pins, except now we don't need to retrieve the data.

Let's create a new function called `addMapPins()`. Make sure to add this to the **ViewDidLoad** method, so that it populates all the map pins when it moves over from the table view. This will make it so that all the locations that are saved are rendered as map pins.

```
/// Add Map Pins in locations array
func addMapPins() {
    for index in 0..
```

Displaying the values in Table View

In order to populate each row in the Table View with each place, we will need to add a few functions to the **Table View Controller**.

First we need to add the **viewWillAppear** function. This function by the system when the view is guaranteed to appear; we need because when we move back from the map view, we want it to refresh the table.

```
override func viewWillAppear(_ animated: Bool) {
    tableView.reloadData()
}
```

Next we need to define how many rows we want in our table. We want one

row per place, so we can just use the count of one of our arrays:

```
/// Returns number of rows
override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
    return addresses.count
}
```

In order to tell our app what information to put in each row, we need the following method:

```
/// Populates rows with text
override func tableView(_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell =
tableView.dequeueReusableCell(withIdentifier: "place",
for: indexPath)
    let address = addresses[(indexPath as
NSIndexPath).row]
    cell.textLabel?.text = address

    return cell
}
```

Adding more features

So now we have a basic app that saves the user's location, or a long press on the map, but it is not perfect.

How do you think we can make it better?

- What if the user quits the app, do the places remain saved?
- What if we try scrolling around the map?
- How can we show a specific location by pressing it in the Table View?

The End

Thank you for coming out to IEEE Carleton's Introductory iOS Workshop. I hope you learned how to start your app development career, and if you have any questions feel free to ask or pop by the Office in ME 3359 for help.

I would appreciate if you would fill out this [feedback form](#) so we can make this workshop even better in the future.