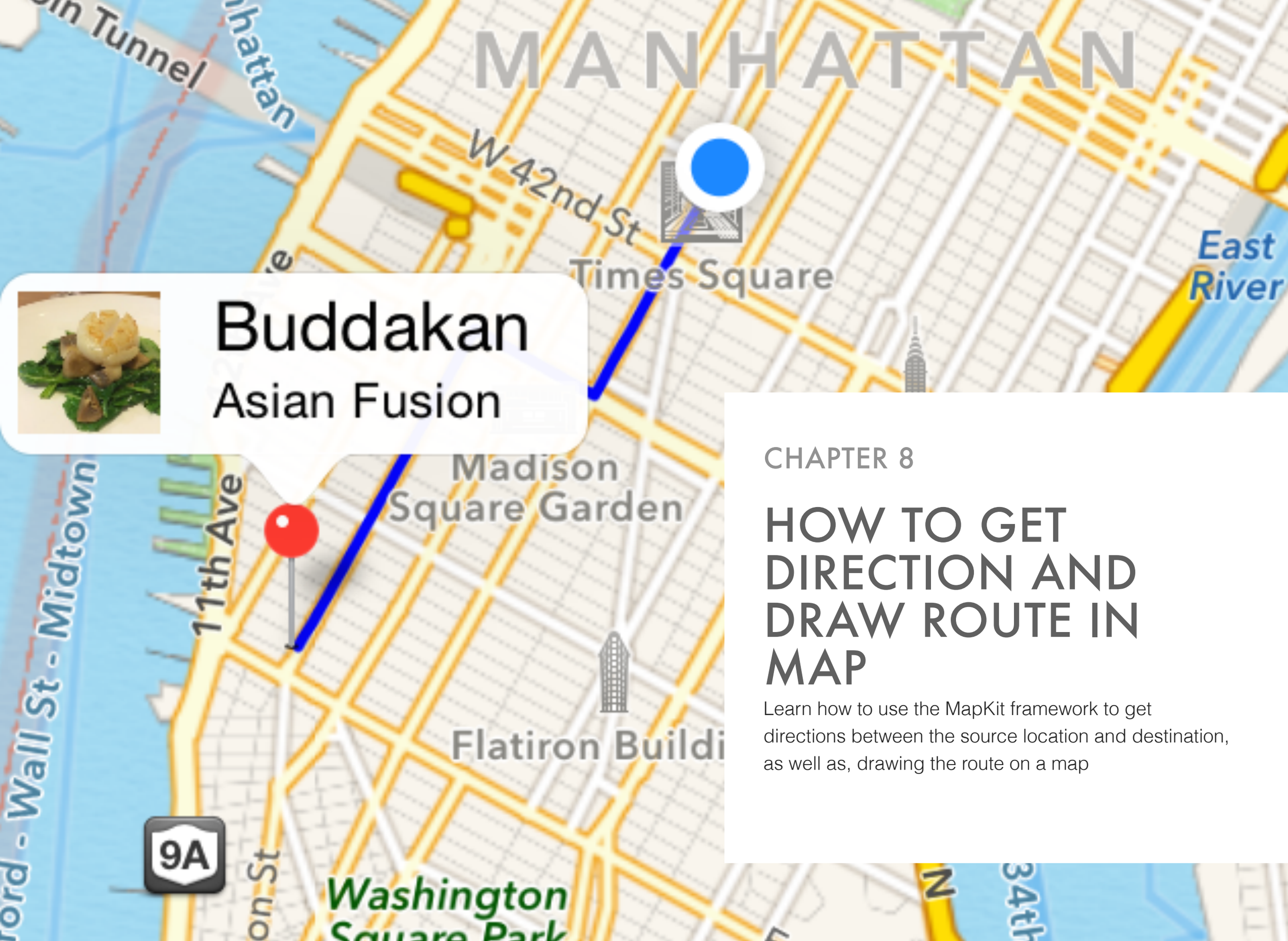


CHAPTER 8

HOW TO GET DIRECTION AND DRAW ROUTE IN MAP

Learn how to use the MapKit framework to get directions between the source location and destination, as well as, drawing the route on a map

THE MKDIRECTIONS API

Since the release of iOS 7 SDK, the MapKit framework includes the MKDirections API which allows iOS developers to access the route-based directions data from Apple's server. Typically, you create an MKDirections instance with the start and end points of a route. The instance then automatically contacts Apple's server and retrieves the route-based data.

You can use the MKDirections API to get both driving and walking directions depending on your preset transport type. If you like, MKDirections can also provide you alternate routes. On top of all that, the API lets you calculate the travel time of a route.

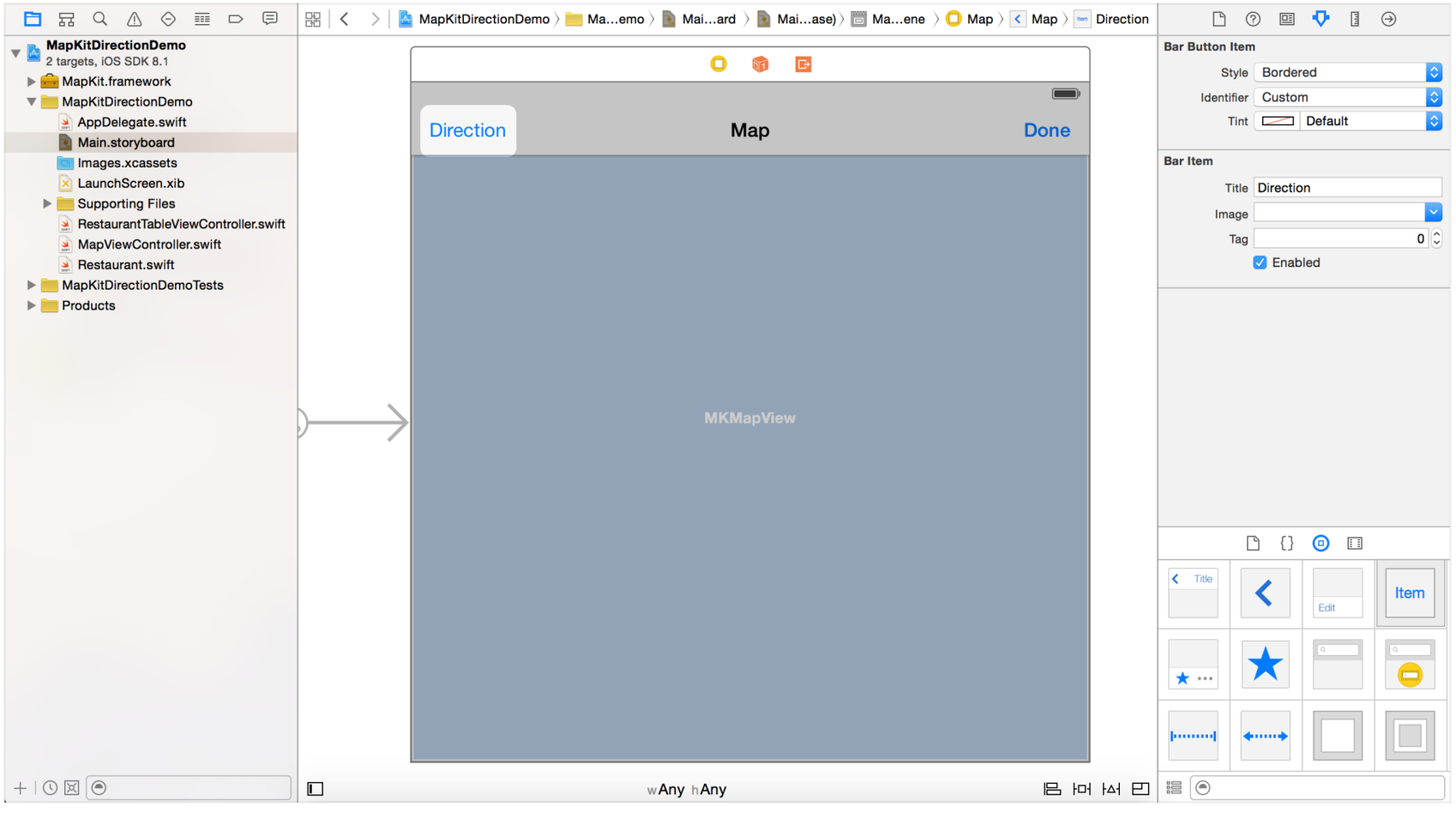
Again we'll build a demo app to see how to utilize the MKDirections API.

SAMPLE ROUTE APP

I have covered the basics of the MapKit framework in the *Beginning iOS 8 Programming with Swift* book, so I expect you have some idea about how MapKit works, and understand how to pin a location on a map. To demonstrate the usage of the MKDirections API, we'll build a sample map app. You can start with this project template (https://www.dropbox.com/s/da8ag58xcjqdfep/MapKitDirectionDemoTemplate.zip?dl=0). If you build the template, you should have an app that shows a list of restaurants. By tapping a restaurant, the app brings you to the map view with the location of the restaurant annotated on the map. That's pretty much the same as what you have implemented in the Food Pin app. We'll enhance the demo app to get the user's current location, and display the directions to the selected restaurant.

ADDING A DIRECTION BUTTON IN THE NAVIGATION BAR

First, open the Xcode project and go to the Main.storyboard. Let's add a Direction button in the navigation bar of the map view controller. Drag a Bar Button Item from the Object Library and add it to the left part of the navigation bar. Set the title of the button to Direction. When this button is tapped, the app will get the user's current location and display the directions to the restaurant.

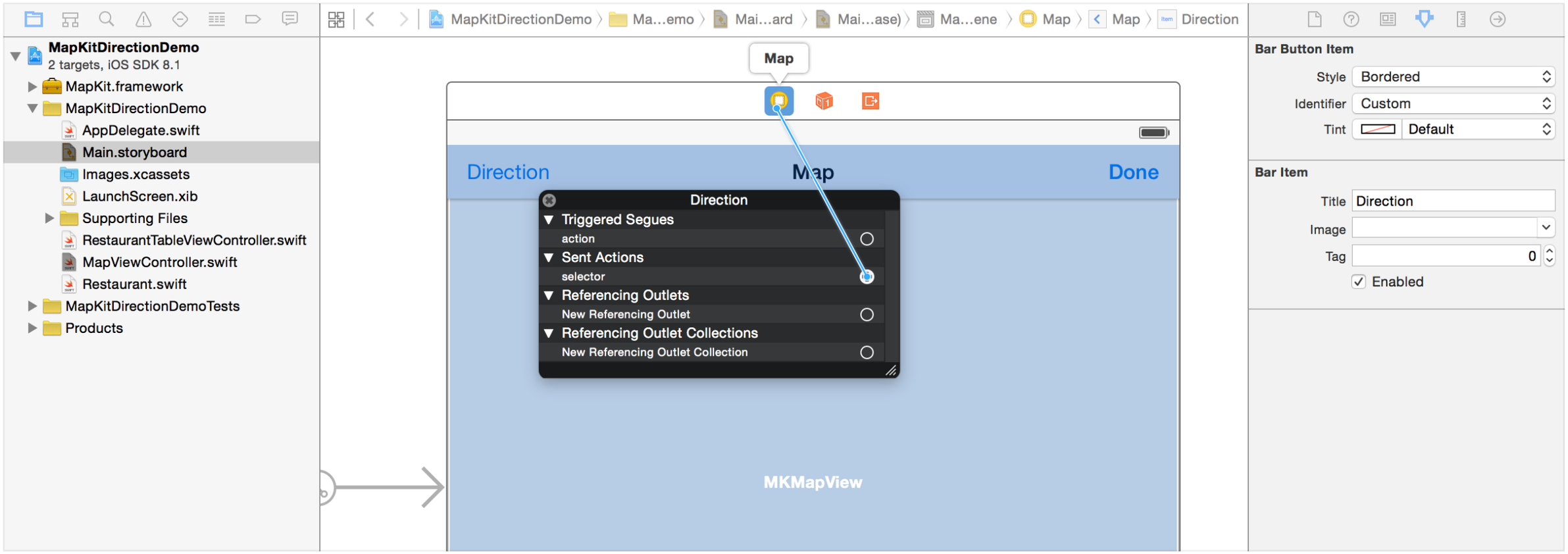


CREATING AN ACTION METHOD FOR THE DIRECTION BUTTON

Next, create an empty action method named *showDirection* in the MapViewController. We'll provide the implementation in later sections.

```
@IBAction func showDirection(sender: AnyObject) {
}
```

In the storyboard, establish a connection between the direction button and the action method. Right-click the Direction button to bring up the context menu. Click the selector circle and drag it to the view controller icon. Select "showDirection" to connect with the action method.



DISPLAYING USER LOCATION ON THE MAP

Since our app is going to display a route from the user's current location to the selected restaurant, we have to enable the map view to show the user's current location. By default, the MKMapView class doesn't display the user's location on the map. You can set the *showsUserLocation* property of the MKMapView class to *true* to enable it. Because the option is set to *true*, the map view uses the built-in Core Location framework to search for the current location and display it on the map.

In the *viewDidLoad* method of the MapViewController class, insert the following line of code:

```
mapView.showsUserLocation = YES
```

If you can't wait to test the app and see how it displays the user location, you can compile and run the app. Select any of the restaurants to bring up the map.

Unfortunately, it doesn't work as expected. If you look into the console, you should find the following error:

```
MapKitDirectionDemo[4041:370059] Trying to start MapKit location updates without prompting for location authorization. Must call -[CLLocationManager requestWhenInUseAuthorization] or -[CLLocationManager requestAlwaysAuthorization] first.
```

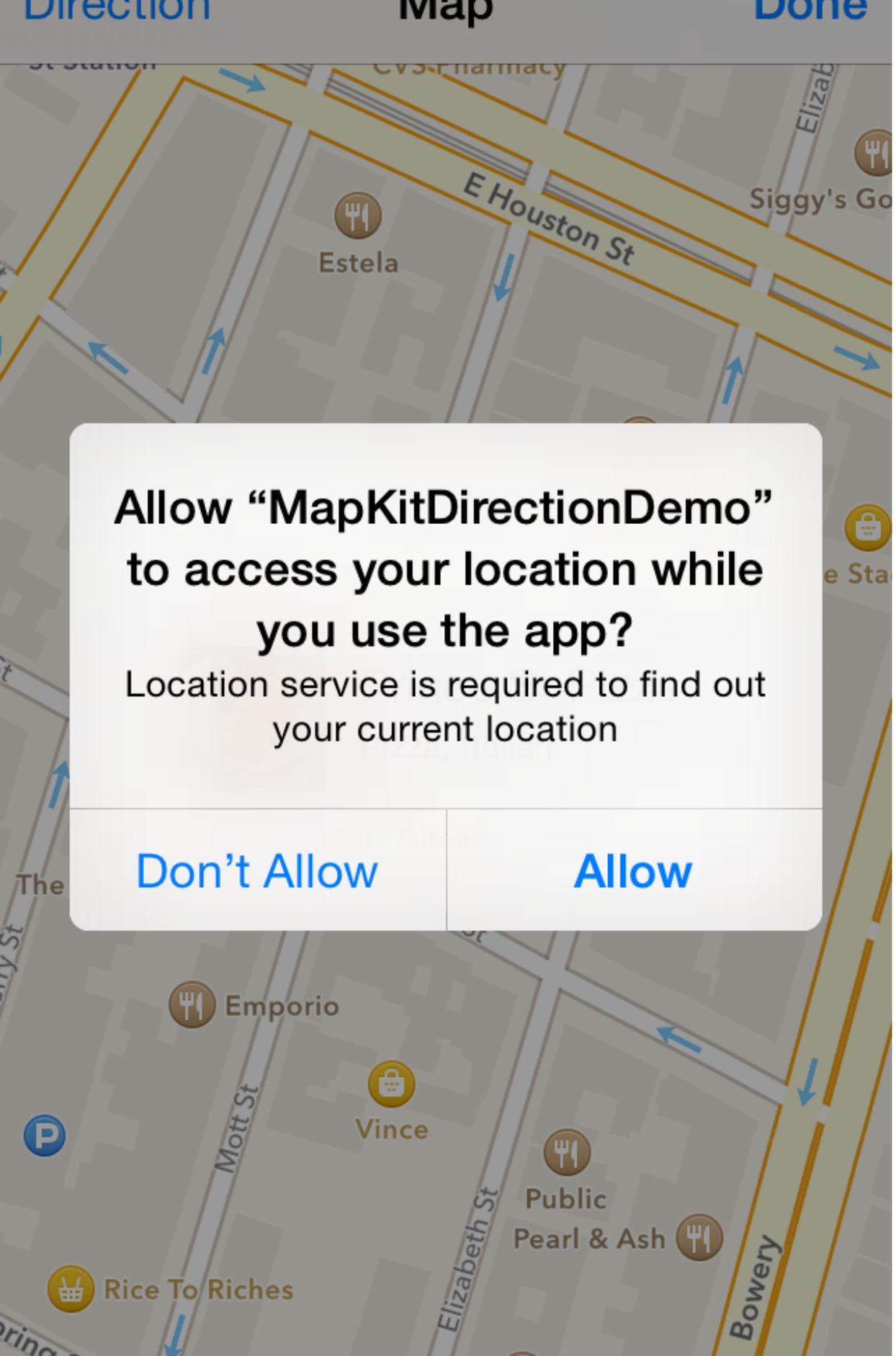
In iOS 8, Core Location introduces a new feature known as Location Authorization. You have to explicitly ask for a user's permission to grant your app location services. Basically, you need to implement these two things to get the location working:

- Request a user's authorization by calling the *requestWhenInUseAuthorization* or *requestAlwaysAuthorization* method of CLLocationManager

- Add a key (NSLocationWhenInUseUsageDescription / NSLocationAlwaysUsageDescription) to your Info.plist

There are two types of authorization: *requestWhenInUseAuthorization* and *requestAlwaysAuthorization.* You use the former if your app only needs location updates when it's in use. The latter is designed for apps that use location services in the background (suspended or terminated). For example, a social app that tracks a user's location requires getting location updates even when it's not running in the foreground. Obviously, the *requestWhenInUseAuthorization* is good enough for our demo app.

Next, you need to add a key to your Info.plist. Depending on the authorization type, you can either add the *NSLocationWhenInUseUsageDescription* or *NSLocationAlwaysUsageDescription* key to the Info.plist. Both keys contain a message telling a user why your app needs location services. For example, you can enter a string like "Location is required to find out your current location."



Okay, let’s modify the code again. First, declare a location manager variable in the MapViewController class:

```
let locationManager = CLLocationManager()
```

Insert the following lines of code in the viewDidLoad method right after super.viewDidLoad():

```
// Request for a user's authorization for location services
locationManager.requestWhenInUseAuthorization()

let status = CLLocationManager.authorizationStatus()
if status == CLAuthorizationStatus.AuthorizedWhenInUse {
   self.mapView.showsUserLocation = true
}
```

The first line of code calls the *requestWhenInUseAuthorization* method. The method first checks the current authorization status. If the user has not yet been asked to authorize location updates, it automatically prompts the user to authorize the use of location services.

Once the user makes a choice, we check the authorization status to see if the user granted permission. If yes, we enable the *showsUserLocation* in the app.

Now run the app again and have a quick test. When you launch the map view, you’ll be prompted to authorize location services. As you can see, the message shown is the one we specified in the NSLocationWhenInUseUsageDescription key.

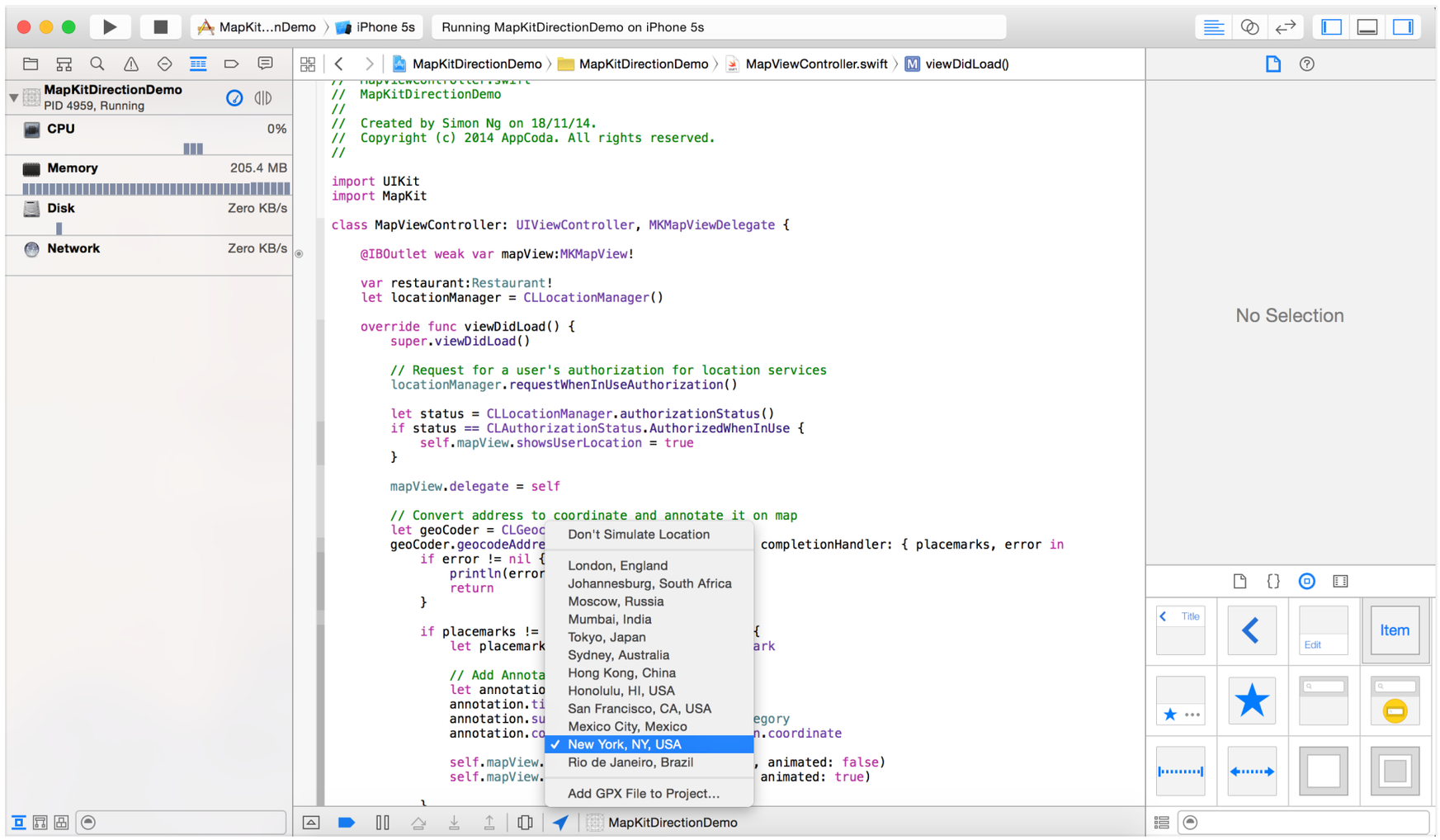
Remember to hit the Allow button to enable the location updates.

TESTING LOCATION USING THE SIMULATOR

Wait! How can we simulate the current location using the built-in simulator? How can you tell the simulator where you are?

There is no way for the simulator to get the current location of your computer. However, the simulator allows you to fake its location. By default, the simulator doesn't simulate the location. You have to enable it manually. While running the app, you can use the "Simulate location" button (arrow button) in the toolbar of the debug area. Xcode comes with a number of preset locations. Just change it to your preferred location (e.g. New York). Alternatively, you can set the default location of your simulator. Just click your scheme > Edit Scheme to bring up the scheme editor. Select the Options tab and set the default location.

Once you set the location, the simulator will display a blue dot in the map which indicates the current user location. If you can't find the blue dot, you may need to zoom the map out. In the simulator, you can hold down the option key to simulate the pinch-in and pinch-out gestures. For details, you can refer to Apple's official document.



USING MKDIRECTIONS API TO GET THE ROUTE INFO

With the user location enabled, we'll compute the route between the current location and the location of the restaurant. First declare a placemark variable in the MapViewController class:

```
var currentPlacemark:CLPlacemark?
```

This variable is used to save the current placemark. In other words, it is the placemark object of the selected restaurant. In the *viewDidLoad* method, locate the following line:

```
let placemark = placemarks[0] as! CLPlacemark
```

Add the following code right below it:

```
self.currentPlacemark = placemark
```

Next we'll implement the *showDirection* method and use the MKDirections API to get the route data. Update the method by using the following code snippet:

```
@IBAction func showDirection(sender: AnyObject) {
let directionRequest = MKDirectionsRequest()

// Set the source and destination of the route
directionRequest.setSource(MKMapItem.mapItemForCurrentLocation())
let destinationPlacemark = MKPlacemark(placemark: currentPlacemark)
directionRequest.setDestination(MKMapItem(placemark: destinationPlacemark))
directionRequest.transportType = MKDirectionsTransportType.Automobile

// Calculate the direction
let directions = MKDirections(request: directionRequest)
directions.calculateDirectionsWithCompletionHandler { (routeResponse, routeError) -> Void in

    if routeError != nil {
        println("Error: \(routeError.localizedDescription)")
    } else {
        let route = routeResponse.routes[0] as! MKRoute
```

```
            self.mapView.addOverlay(route.polyline, level: MKOverlayLevel.AboveRoads)
        }
    }
}
```

To request directions, we first create an instance of MKDirectionsRequest. The class is used to store the source and destination of a route. There are a few optional parameters you can configure such as transport type, alternate routes, etc. In the above code, we just set the source, destination and transport type while using default values for the rest of the options. The starting point is set to the user's current location. You can use MKMapItem.mapItemForCurrentLocation to retrieve the current location. The end point of the route is set to the destination of the selected restaurant. The transport type is set to automobile.

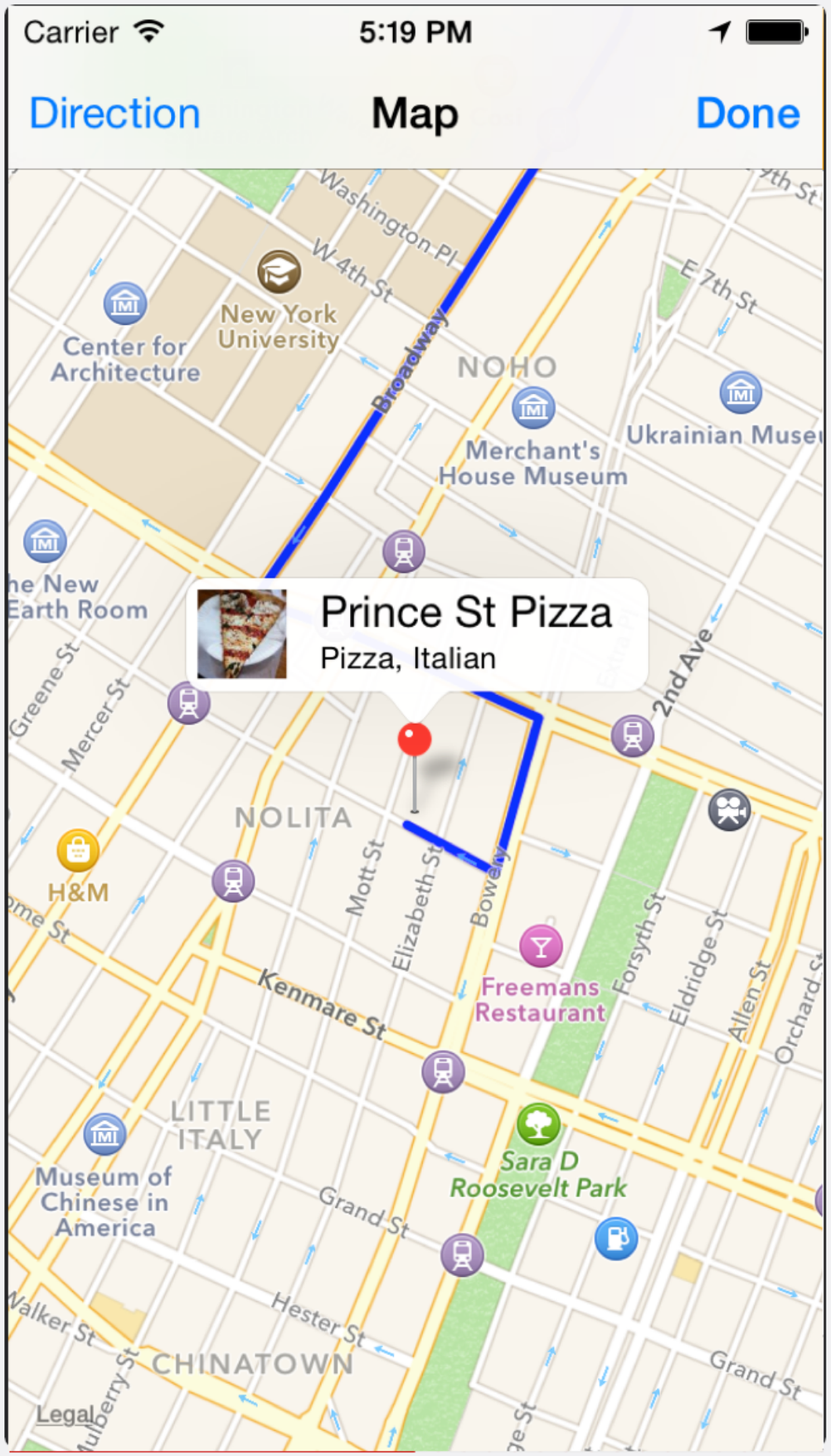
With the MKDirectionsRequest created, we instantiate the MKDirections object and call the *calculateDirectionsWithCompletionHandler* method. The method initiates an asynchronous request for directions and calls your completion handler when the request is completed. The MKDirections object simply passes your request to the Apple servers and asks for route-based directions data. Once the request completes, the completion handler is called. The route information returned by the Apple servers is returned as an MKDirectionsResponse object. MKDirectionsResponse provides a container for saving the route information so that the routes are saved in a *routes* property.

In the completion handler block, we retrieve the first MKRoute object from the route response (MKDirectionsResponse). By default, only one route is returned. Apple may return multiple routes if the *requestsAlternateRoutes* property of the MKDirectionsRequest object is enabled. Because we didn't enable the alternate route option, we just pick the first route.

With the route, we add it to the map by calling the *addOverlay* method of the MKMapView class. The detailed route geometry (i.e. route.polyline) is represented by an MKPolyline object. The *addOverlay* method is used to add an MKPolyline object to the existing map view. Optionally, we configure the map view to overlay the route above roadways but below map labels or point-of-interest icons.

That's how you construct a direction request and overlay a route on map. If you run the app now, you will not see a route when the Direction button is tapped. There is still one thing left. We need to implement the *rendererForOverlay* method which actually draws the route:

```
func mapView(mapView: MKMapView!, rendererForOverlay overlay: MKOverlay!) -> MKOverlayRenderer! {
    let renderer = MKPolylineRenderer(overlay: overlay)
    renderer.strokeColor = UIColor.blueColor()
    renderer.lineWidth = 3.0

    return renderer
```




```
}
```

In the method, we create a *MKPolylineRenderer* which provides the visual representation for the specified *MKPolyline* overlay object. Here the overlay object is the one we added earlier.

The renderer object provides various properties to control the appearance of the route path. We simply change the stroke color and line width.

Okay, let's run the app again and you should be able to see the route after pressing the Direction button. If you can't view the path, remember to check if you set the simulated location to New York.

SCALE THE MAP TO MAKE THE ROUTE FIT PERFECTLY

You should notice a problem with the current implementation. The demo app does indeed draw the route on the map, but you may need to zoom the map out manually in order to show the route. Could we scale the map automatically?

You can use the boundingMapRect property of the polyline to determine the smallest rectangle that completely encompasses the overlay and changes the visible region of the map view.

Insert the following lines of code in the *showDirection* method:

```
let rect = route.polyline.boundingMapRect
self.mapView.setRegion(MKCoordinateRegionForMapRect(rect), animated: true)
```

And place them right after the following line of code:

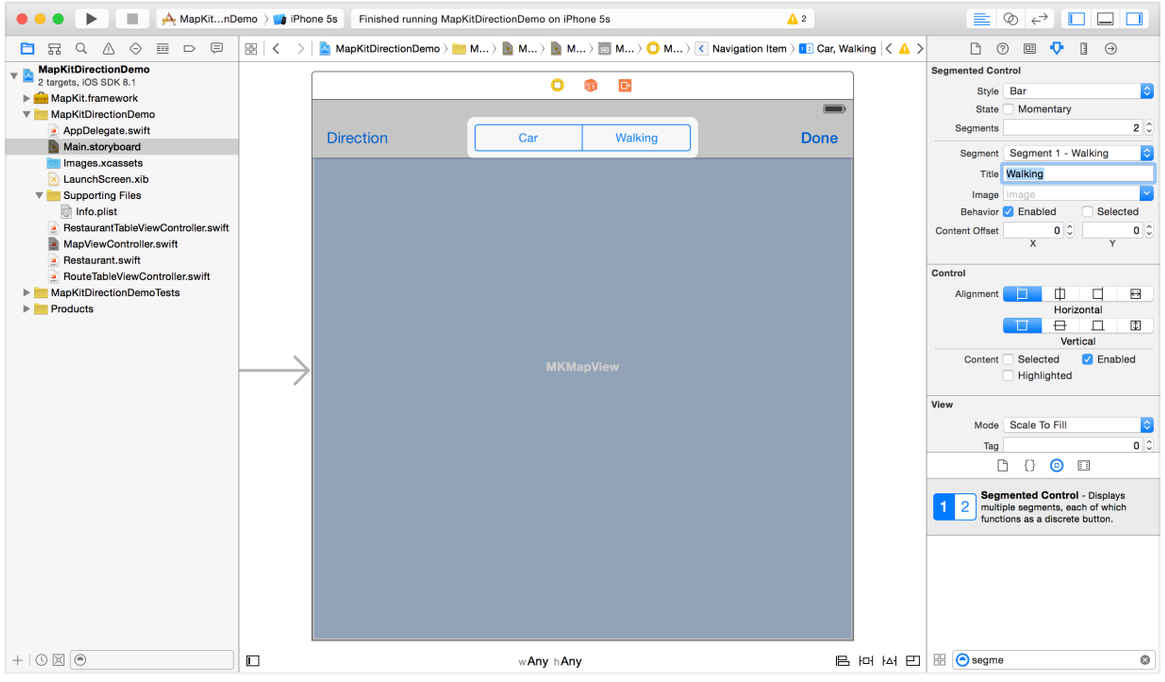
```
self.mapView.addOverlay(route.polyline, level: MKOverlayLevel.AboveRoads)
```

Compile and run the app again. The map should now scale automatically to display the route within the screen real estate.

USING SEGMENTED CONTROL

Presently, the app only provides route information for an automobile. Wouldn't it be great if the app supported walking directions? We'll add a segmented control in the app such that users can choose between driving and walking directions. A segmented control is a horizontal control made of multiple segments. Each segment of the control functions like a button.

Now go to the storyboard. Drag a segmented control from the Object Library and place it in the navigation bar of the map view controller. Select the segmented control and go to the Attributes Inspector. Change the title of the first item to "Car" and the second item to "Walking."



Next, go to the MapViewController.swift. Declare an outlet variable for the segmented control:

```
@IBOutlet weak var segmentedControl:UISegmentedControl!
```

Go back to the storyboard and connect the segmented control with the outlet variable.

In the viewDidLoad method of the MapViewController.swift, put this line of code right after super.viewDidLoad():

```
segmentedControl.hidden = true
```

Obviously, you want to display the control only when a user taps the direction button. This is why we hide it when the view controller is first loaded up.

Next, declare a new instance variable in the MapViewController class:

```
var currentTransportType = MKDirectionsTransportType.Automobile
```

The variable indicates the selected transport type. By default, it is set to automobile (i.e. car).

Due to the use of this variable, we have to change the following line of code in the showDirection method:

```
directionRequest.transportType =
MKDirectionsTransportType.Automobile
```

Simply replace *MKDirectionsTransportType.Automobile* with *currentTransportType*.

Okay, you've got everything in place. But how can you detect the user's selection of a segmented control? When a user presses one of the segments, the control sends the *ValueChanged* event. So what you need is to register the event and perform the corresponding action when the event is triggered.

You can register the event by control-dragging the segmented control's Value Changed event from the Connections inspector to the action method.

Since you're now an intermediate programmer, let's see how you can register the event by writing code. Typically, you register the target-action methods for a segmented control like this:

```
segmentedControl.addTarget(self, action: "showDirection:", forControlEvents: .ValueChanged)
```

You use the *addTarget* method to register the .ValueChanged event. When the event is triggered, we instruct the control to call the *showDirection* method of the current object (i.e. MapViewController).

Since we need to check the selected segment, insert the following code snippet at the very beginning of the *showDirection* method:

```
if segmentedControl.selectedSegmentIndex == 0 {
    currentTransportType = MKDirectionsTransportType.Automobile
} else {
    currentTransportType = MKDirectionsTransportType.Walking
}
segmentedControl.hidden = false
```

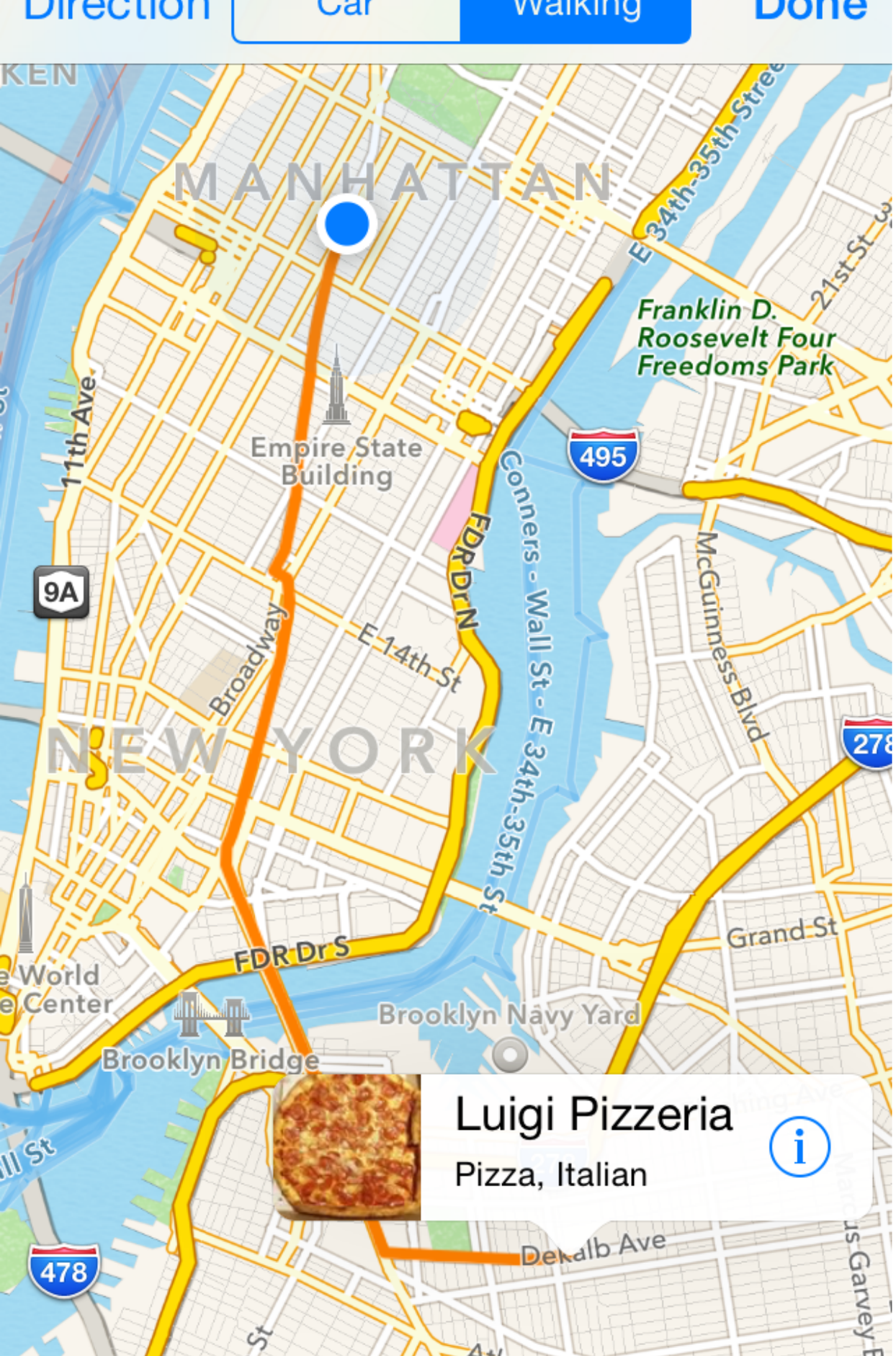
The *selectedSegmentedIndex* property of the segmented control indicates the index of the selected segment. If the first segment (i.e. Car) is selected, we set the current transport type to automobile. Otherwise, it is set to walking. We also unhide the segmented control.

Lastly, insert the following line of code (highlighted in yellow) in the *calculateDirectionsWithCompletionHandler* closure:

```
directions.calculateDirectionsWithCompletionHandler { (routeResponse, routeError) -> Void in

        if routeError != nil {
            println("Error: \(routeError.localizedDescription)")
        } else {
            let route = routeResponse.routes[0] as! MKRoute
            self.mapView.removeOverlays(self.mapView.overlays)
            self.mapView.addOverlay(route.polyline, level: MKOverlayLevel.AboveRoads)

            // Scale the map
            let rect = route.polyline.boundingMapRect

self.mapView.setRegion(MKCoordinateRegionForMapRect(rect), animated: true)
        }
    }
```

The line of code simply asks the map view to remove all the overlays. This is to avoid both "Car" and "Walk" routes overlapping with each other.

You can now test the app. In the map view, tap the Direction button and the segmented control should appear. You're free to select the Walking segment to display the walking directions.



For now, both types of routes are shown in blue. You can make a minor change in the rendererForOverlay method of the MapViewController class to display a different color. Simply change this line of code:

```
renderer.strokeColor = (currentTransportType == .Automobile) ?
UIColor.blueColor() : UIColor.orangeColor()
```

We use blue color for the “Car” route and orange color for the “Walking” route.

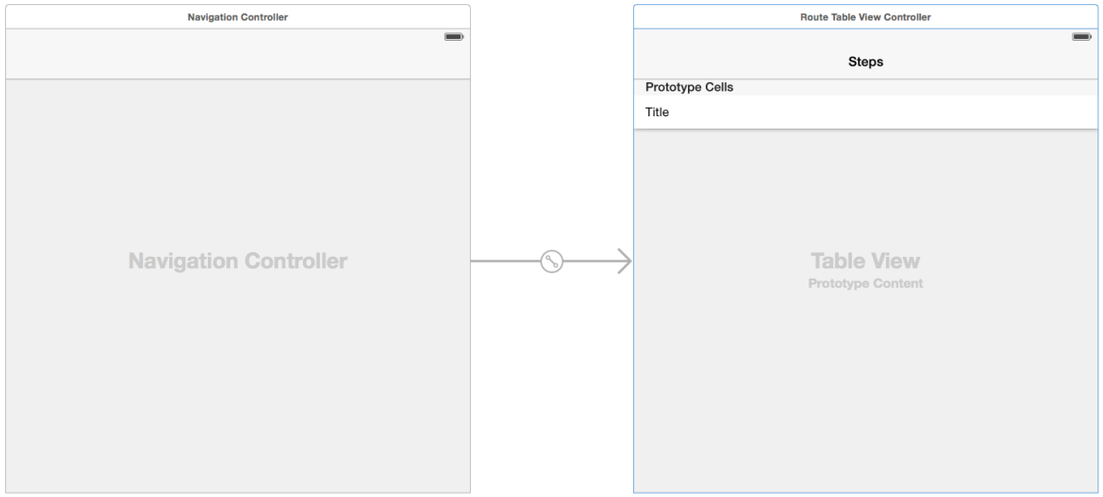
After the change, run the app again. When walking is selected, the route is displayed in orange.

SHOWING ROUTE STEPS

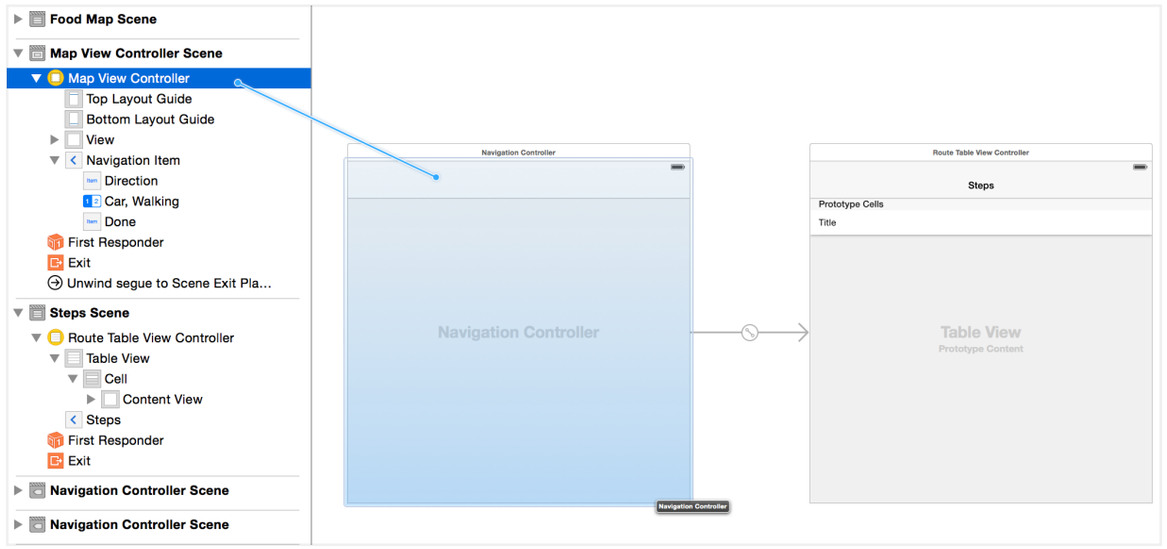
Now that you know how to display a route in a map, wouldn't it be great if you could provide detailed driving (or walking) directions for your users? The MKRoute object provides a property called **steps**, which contains an array of MKRouteStep objects. An MKRouteStep object represents one part of an overall route. It also provides a single instruction that would need to be followed by the user.

Okay, let's tweak the demo. When someone taps the annotation, the app will display the detailed driving/walking instructions.

First, add a table view controller to the storyboard and embed it in a navigation controller. Set the title of the navigation bar to "Steps" and the identifier of the prototype cell as to "Cell".



Next, connect the map view controller with the new navigation controller using a segue. In the Document Outline of Interface Builder, control-drag the map view controller to the navigation controller. Select "show" for the segue type and set the segue's identifier to "showSteps".



The UI design is ready. Now create a new class file using the Cocoa Touch class template. Name it RouteTableViewController and make it a subclass of UITableViewController.

Once the class is created, go back to the storyboard. Select the Steps table view controller. Under the Identity inspector, set the custom class to RouteTableViewController.

Apparently, there are two questions in your head:

- How can we get the detailed steps from the route?

- How do we know if a user touches the annotation in a map?

As I mentioned earlier, the *steps* property of an MKRoute object contains an array of MKRouteStep objects. Each MKRouteStep object comes with an instructions property that stores the written instructions (e.g. Turn right onto Charles St) for following the path of a particular step. So all we need to do is loop through all the MKRouteStep objects to display the written instructions in the Steps table view.

Similar to a table view, the MKAnnotationView provides an optional accessory view displayed on the right side of a standard callout bubble. Once you create the accessory view, the *calloutAccessoryControlTapped* method of your map view's delegate will be called when a user taps the accessory view.

Now that you should have a better idea of the implementation, let's continue to develop the app.

First open the RouteTableViewController.swift file and declare an instance variable:

```
var routeSteps = [MKRouteStep]()
```

This variable is used for storing an array of MKRouteStep object of a selected route.

Replace the method of table view data source with the following:

```
override func numberOfSectionsInTableView(tableView: UITableView) ->
Int {
    // Return the number of sections.
    return 1
}

override func tableView(tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
    // Return the number of rows in the section.
    return routeSteps.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath) as! UITableViewCell

    // Configure the cell...
    cell.textLabel?.text = routeSteps[indexPath.row].instructions

    return cell
}
```

The above code is very straightforward. We simply display the written instructions of the route steps in the table view.

Next, open MapViewController.swift. We're going to add a few lines of code to handle the finishing touches of annotation.

At the very beginning of the class, declare a new variable to store the current route:

```
var currentRoute:MKRoute?
```

In the *viewForAnnotation* method, insert the following line of code before "return annotationView":

```
annotationView.rightCalloutAccessoryView =
UIButton.buttonWithType(UIButtonType.DetailDisclosure) as! UIView
```

Here we add a detail disclosure button to the right side of an annotation.

To handle a touch, we implement the *calloutAccessoryControlTapped* method like this:

```
func mapView(mapView: MKMapView!, annotationView view:
MKAnnotationView!, calloutAccessoryControlTapped control: UIControl!)
{
    self.performSegueWithIdentifier("showSteps", sender: view)
}
```

In iOS, you're allowed to trigger a segue programmatically by calling the *performSegueWithIdentifier* method. Earlier we created a segue between the map view controller and the navigation controller and set the segue's identifier to "showSteps." The app will navigate to the Steps table view controller when the above *performSegueWithIdentifier* method is called.

Lastly, we have to pass the current route steps to the RouteTableViewController class.

In the body of the *calculateDirectionsWithCompletionHandler* closure, insert a line of code (highlighted in yellow) to update the current route:

```
directions.calculateDirectionsWithCompletionHandler { (routeResponse,
routeError) -> Void in

    if routeError != nil {
        println("Error: \(routeError.localizedDescription)")
    } else {
        let route = routeResponse.routes[0] as! MKRoute
        self.currentRoute = route
        self.mapView.removeOverlays(self.mapView.overlays)
        self.mapView.addOverlay(route.polyline, level:
MKOverlayLevel.AboveRoads)

        // Scale the map
        let rect = route.polyline.boundingMapRect
        self.mapView.setRegion(MKCoordinateRegionForMapRect(rect),
animated: true)
    }
}
```

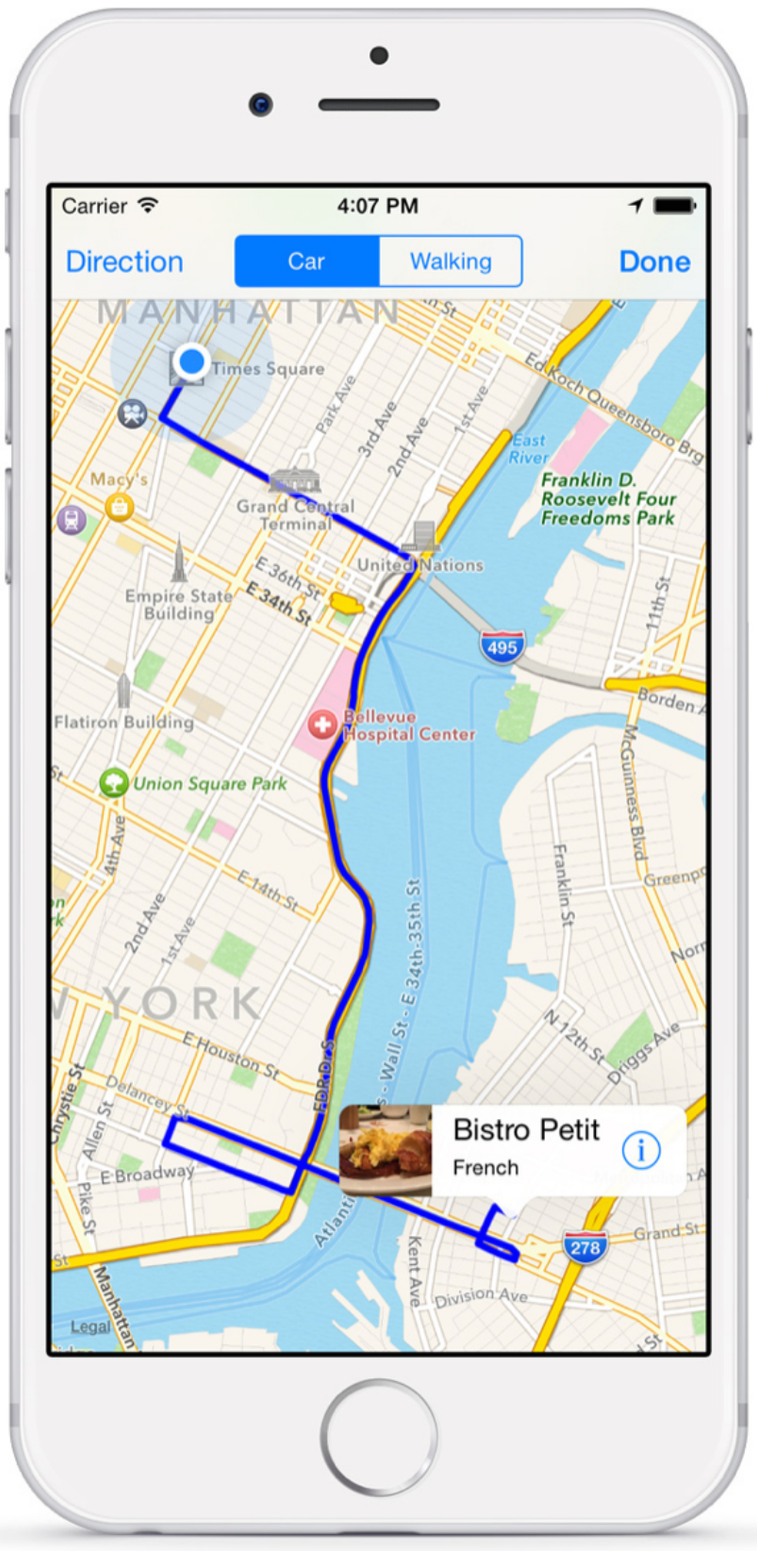
To pass the route steps to the RouteTableViewController, implement the *prepareForSegue* method like this:

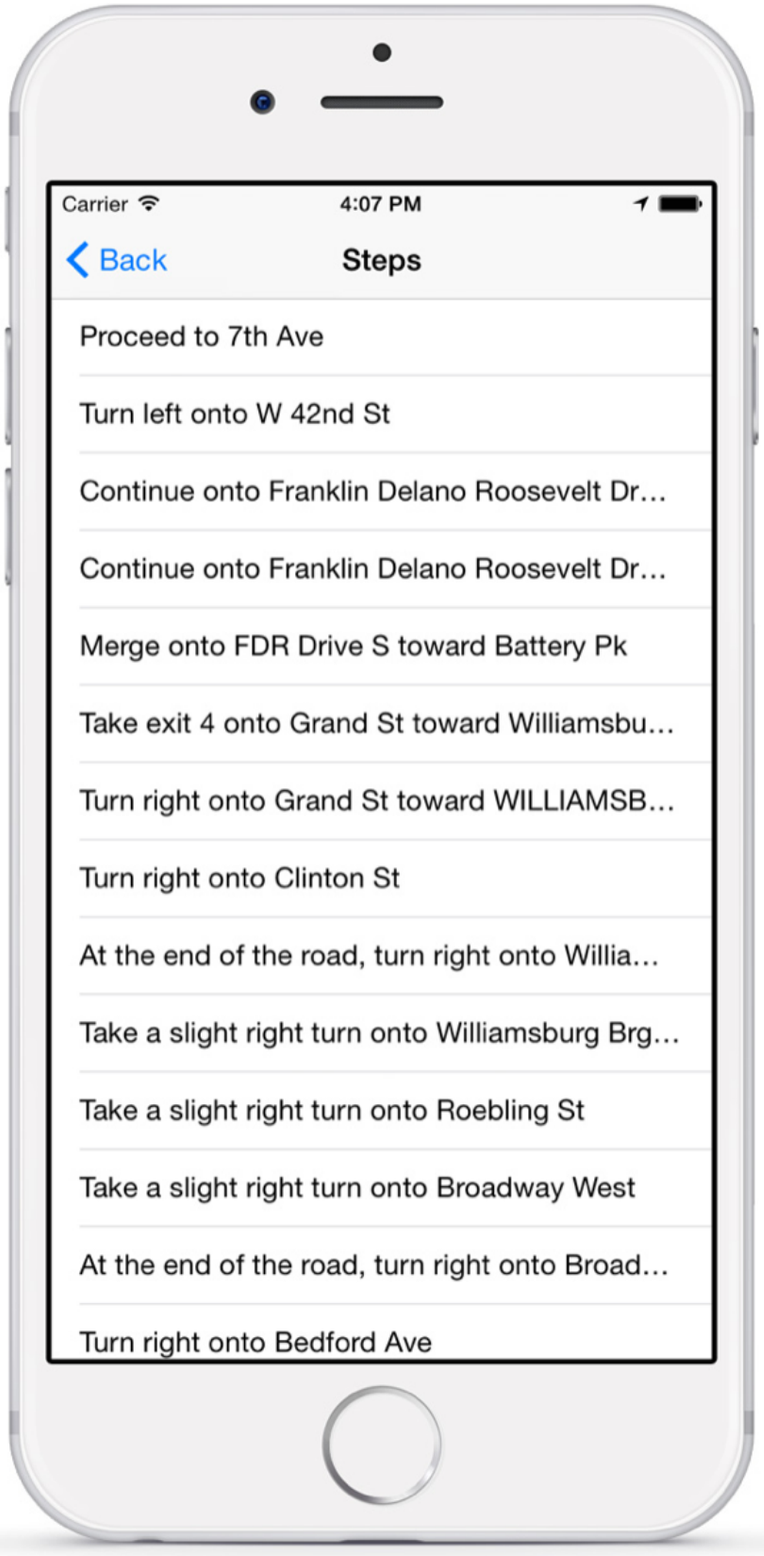
```
override func prepareForSegue(segue: UIStoryboardSegue, sender:
AnyObject?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.
    if segue.identifier == "showSteps" {
        let destinationController = segue.destinationViewController as!
UINavigationController
        let routeTableViewController =
destinationController.childViewControllers.first as!
RouteTableViewController
        if let steps = currentRoute?.steps as? [MKRouteStep] {
            routeTableViewController.routeSteps = steps
        }
    }
}
```

The above code snippet should be very familiar to you. We first get the destination controller, which is a navigation controller. From the navigation controller, we retrieve the RouteTableViewController object and pass it the route steps.

The app's now ready to run. When you tap the annotation on the map, the app shows you a list of steps to follow.

For your reference, you can download the complete Xcode project from https://www.dropbox.com/s/kzkyokepakkzx0i/MapKitDirectionDemo.zip?dl=0.


Carrier
4:07 PM
Direction
Car
Walking
Done
MANHATTAN
Times Square
Macy's
Grand Central Terminal
Empire State Building
Flatiron Building
Union Square Park
Bellevue Hospital Center
United Nations
East River
Franklin D. Roosevelt Four Freedoms Park
Ed Koch Queensboro Brg
Park Ave
3rd Ave
2nd Ave
1st Ave
E 36th St
E 34th St
11th St
495
Borden
McGuinness Blvd
Franklin St
Greenpoint
4th Ave
2nd Ave
1st Ave
E 14th St
YORK
E Houston St
Delancey St
Chrystie St
Allen St
Pike St
E Broadway
Bistro Petit
French
Atlantic
Wall St - E 34th-35th St
N 12th St
Driggs Ave
Grand St
278
Kent Ave
Division Ave
Manhattan
Legal


Carrier
4:07 PM
Back
Steps
Proceed to 7th Ave
Turn left onto W 42nd St
Continue onto Franklin Delano Roosevelt Dr…
Continue onto Franklin Delano Roosevelt Dr…
Merge onto FDR Drive S toward Battery Pk
Take exit 4 onto Grand St toward Williamsbu…
Turn right onto Grand St toward WILLIAMSB…
Turn right onto Clinton St
At the end of the road, turn right onto Willia…
Take a slight right turn onto Williamsburg Brg…
Take a slight right turn onto Roebling St
Take a slight right turn onto Broadway West
At the end of the road, turn right onto Broad…
Turn right onto Bedford Ave