How to Get Direction and Draw Route on Maps

Since the release of the 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.

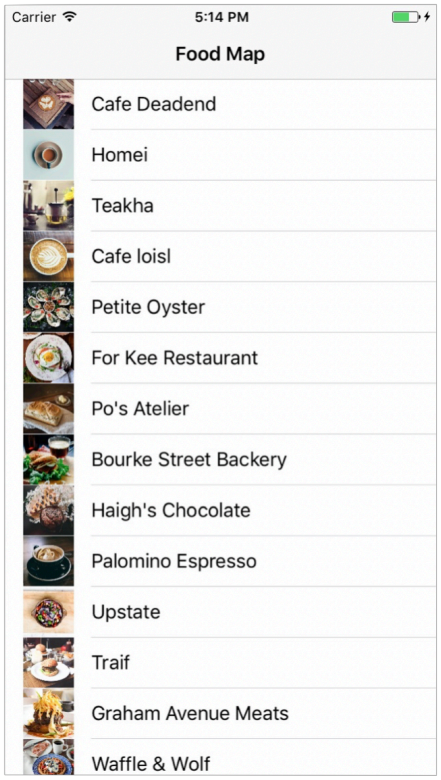
You can use the MKDirections API to get both driving and walking directions depending on your preset transport type. If you like, MKDirections can also provide you alternate routes. On top of all that, the API lets you calculate the travel time of a route.

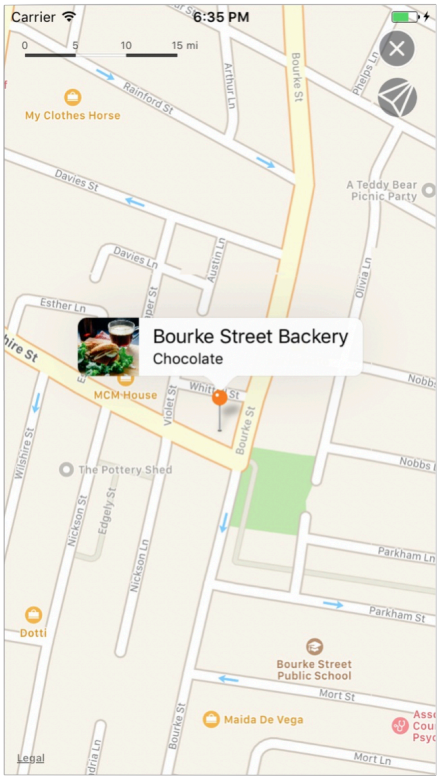
Again we'll build a demo app to see how to utilize the MKDirections API.

Sample Route App

We already 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 simple map app. You can start with the project template (MapKitDirectionStarter.zip).

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. We'll enhance the starter app to get the user's current location, and display the directions to the selected restaurant.





There is one thing we have to point out. If you look into the MapViewController class, you will find these lines of code:

```
if #available(iOS 9.0, *) {
    mapView.showsCompass = true
    mapView.showsScale = true
    mapView.showsTraffic = true
}
```

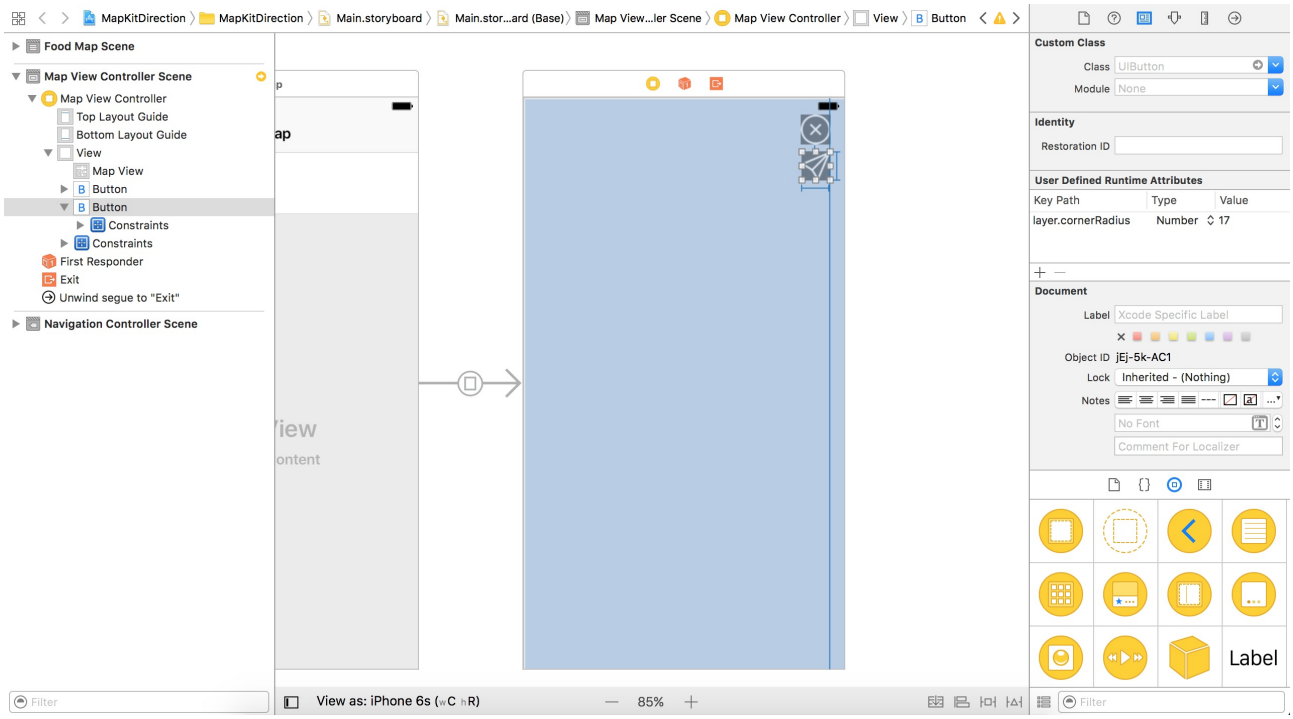
The current project is configured to run on iOS 8.4 or up. However, some of the APIs are available in the iOS 9 SDK or later. For example, the showsCompass , showsScale and showsTraffic properties are not available until the release of iOS 9. Similar to this project, if your app is going to support iOS 8, 9 and 10, you will need to check the OS version before calling some newer APIs. Otherwise, this will cause errors when the app runs on older versions of iOS.

From Swift 2 and onwards, the programming language has a built-in support for API availability checking. You can easily define an

availability condition such that the block of code will only be executed on certain iOS versions. You use the #available keyword in a if statement. In the availability condition, you specify the OS versions (e.g. iOS 9, OSX 10.10) you want to verify. The asterisk (*) is required and indicates that the if clause is executed on the minimum deployment target and any other versions of OS. In the above example, we will execute the code block only if the device is running on iOS 9 (or up).

Creating an Action Method for the Direction Button

Now, open the Xcode project and go to Main.storyboard . The starter project already comes with the direction button, but it is not working yet.



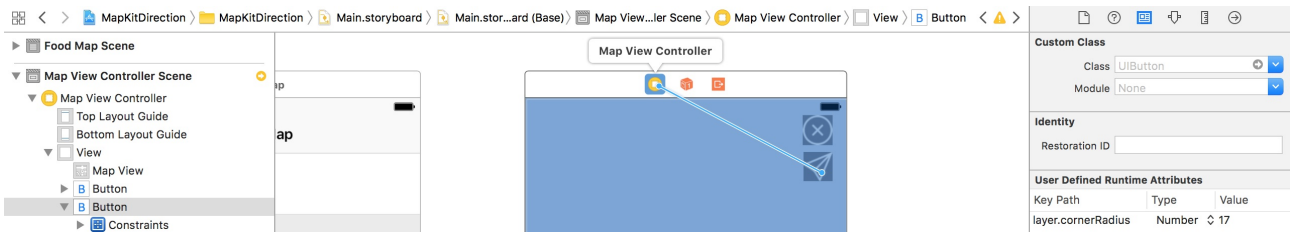
So what we are going to do is to implement this button. When a user taps the button, it shows the user's current location, and displays the directions to the selected restaurant.

The map view controller has been associated with the MapViewController class. Now, create an empty action method named showDirection in the class. We'll provide the implementation in the later section.

```
@IBAction func showDirection(_ sender: Any) {
}
```

In the storyboard, establish a connection between the Direction button and the action method. Control-drag from the Direction button to the view controller icon in the dock. Select

showDirection: to connect with the action method.



Displaying the User Location on Maps

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 = true
```

You can compile and run the app. Select any of the restaurants to bring up the map. It doesn't work as expected yet. The app doesn't show your current location.

Starting from 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 location updates even if it's not running in the foreground. Obviously,

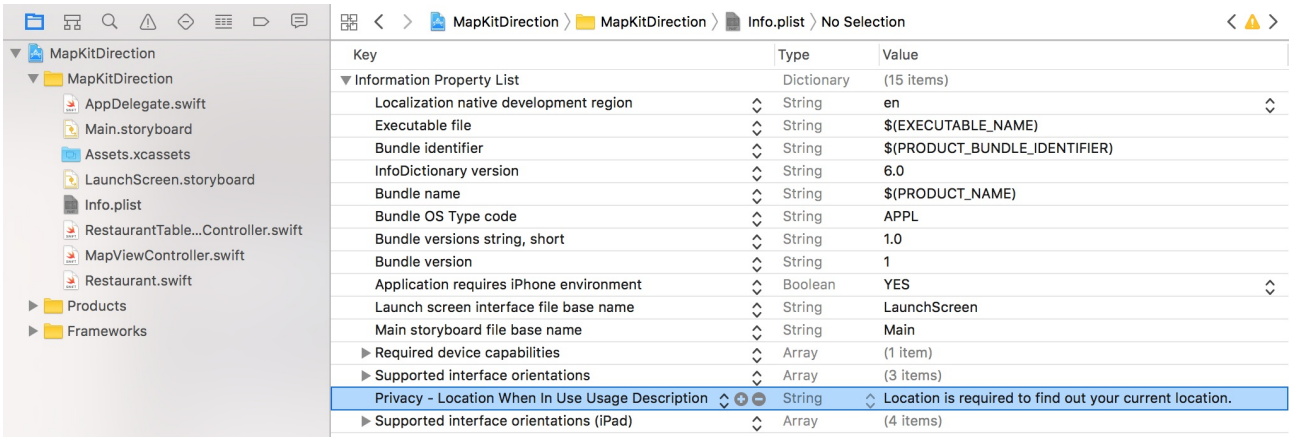
requestWhenInUseAuthorization is good enough for our app.

To do that, you will need to add a key to your Info.plist . Depending on the authorization type, you can either add the NSLocationWhenInUseUsageDescription or NSLocationAlwaysUsageDescription key to Info.plist . Both keys contain a message telling a user why your app needs location services.

In this project, let's add the NSLocationWhenInUseUsageDescription key in Info.plist . Select the file and right click any blank. Choose Add Row in the popover menu. For the key, set it to

Privacy - Location When in Use Usage Description , which is actually the

NSLocationWhenInUseUsageDescription key. For the value, enter Location is required to find out your current location.



Now we are ready to modify the code again. First, declare a location manager variable in the MapViewController class:

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

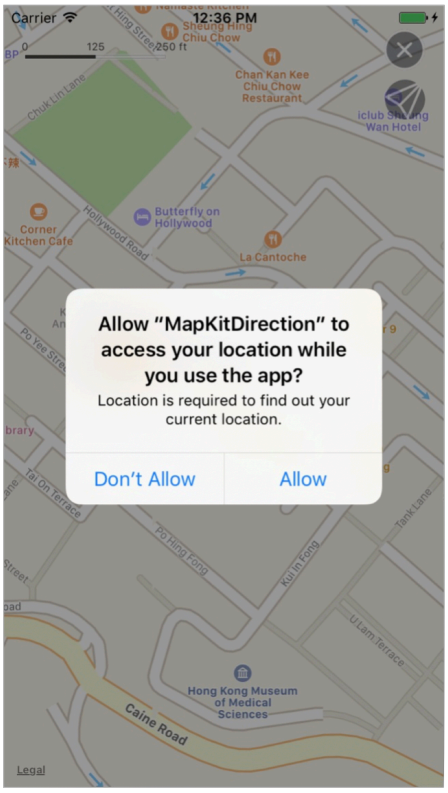
Insert the following lines of code in the viewDidLoad method right after super.viewDidLoad() :

```
// Request for a user's authorization for location services
locationManager.requestWhenInUseAuthorization()
let status = CLLocationManager.authorizationStatus()
if status == CLAuthorizationStatus.authorizedWhenInUse {
    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 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.

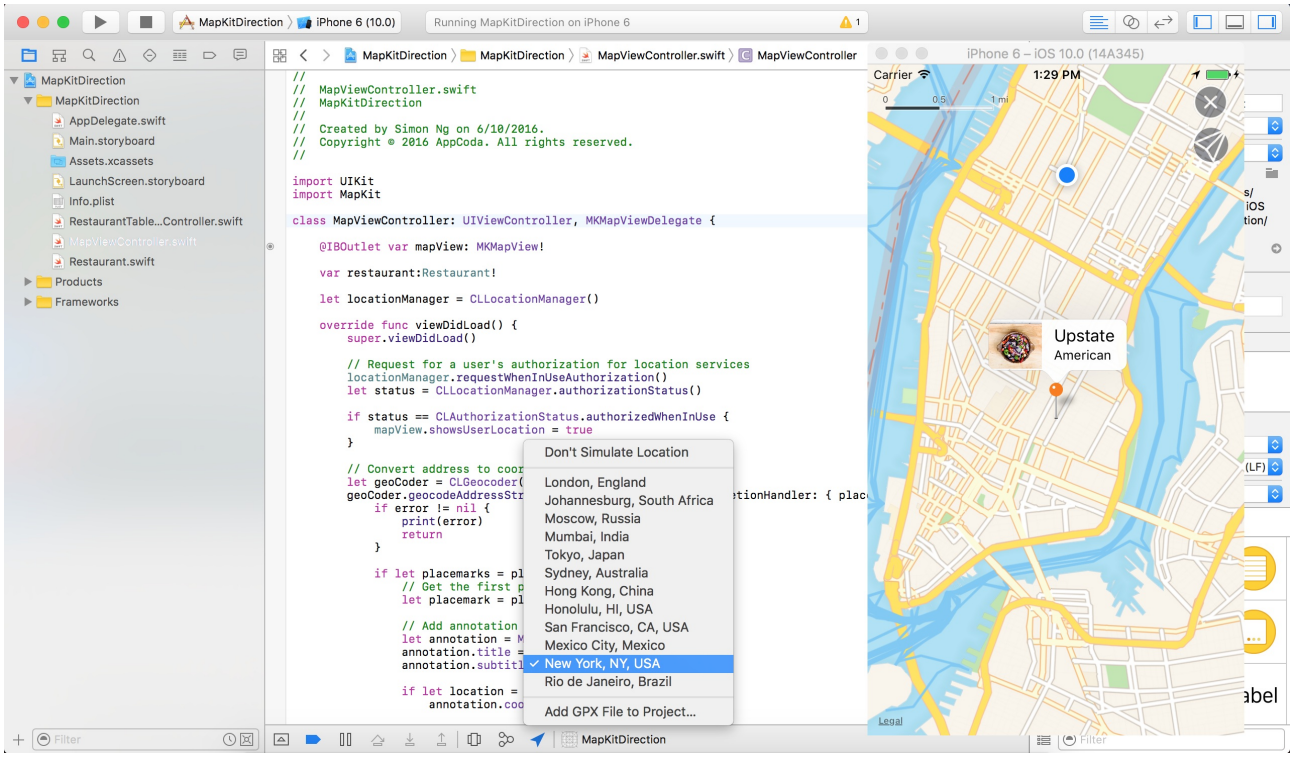


Testing Location Using the Simulator

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. Moscow). 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 on the map, simply zoom out. In the simulator, you can hold down the option key to simulate the pinch-in and pinch-out gestures.



Using MKDirections API to Get the Route info

With the user location enabled, we move on to 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. A placemark in iOS stores information such as country, state, city and street address for a specific latitude and longitude.

In the starter project, we already retrieve the placemark object of the selected restaurant. In the viewDidLoad method, you should be able to locate the following line:

```
let placemark = placemarks[0]
```

Next, add the following code right below it to set the value of currentPlacemark :

```
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: Any) {

    guard let currentPlacemark = currentPlacemark else {
        return
    }
    let directionRequest = MKDirectionsRequest()
    // Set the source and destination of the route
    directionRequest.source = MKMapItem.forCurrentLocation()
    let destinationPlacemark = MKPlacemark(placemark: currentPlacemark)
    directionRequest.destination = MKMapItem(placemark: destinationPlacemark)
    directionRequest.transportType = MKDirectionsTransportType.automobile
    // Calculate the direction
```

```
    let directions = MKDirections(request: directionRequest)
    directions.calculate { (routeResponse, routeError) -> Void in
        guard let routeResponse = routeResponse else {
            if let routeError = routeError {
                print("Error: \(routeError)")
            }
            return

        }
        let route = routeResponse.routes[0]
        self.mapView.add(route.polyline, level: MKOverlayLevel.aboveRoads)
    }
}
```

At the beginning of the method, we make sure if currentPlacemark contains a value using a guard statement. Otherwise, we just skip everything.

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. We 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 object created, we instantiate an MKDirections object and call the calculate(completionHandler:) 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 the routes property.

In the completion handler block, we first check if the route response contains a value. Otherwise, we just print the error. If we can successfully get the route response, we retrieve the first MKRoute object. 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 add(_:level:) method of the MKMapView class. The detailed route geometry (i.e. route.polyline) is represented by an MKPolyline object. The add(_:level:) 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.

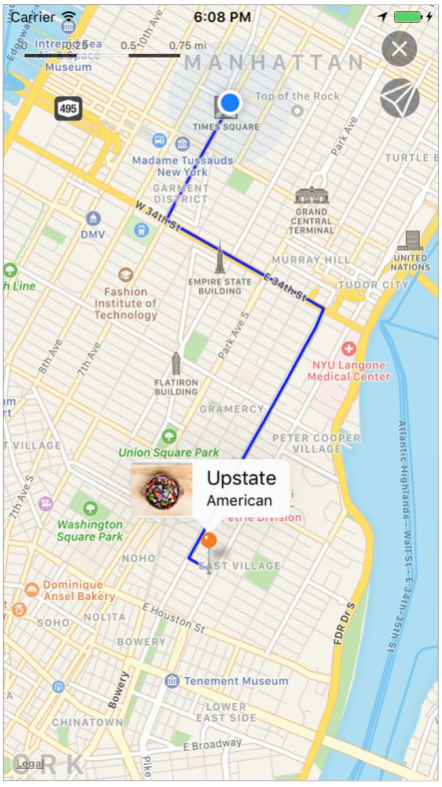
That's how you construct a direction request and overlay a route on map. If you run the app now, you will not see a route when the Direction button is tapped. There is still one thing left.

We need to implement the mapView(_:rendererFor:) method which actually draws the route:

```
func mapView(_ mapView: MKMapView, rendererFor overlay: MKOverlay) ->
MKOverlayRenderer {
    let renderer = MKPolylineRenderer(overlay: overlay)
    renderer.strokeColor = UIColor.blue
    renderer.lineWidth = 3.0
    return renderer
}
```

In the method, we create an MKPolylineRenderer object 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 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:

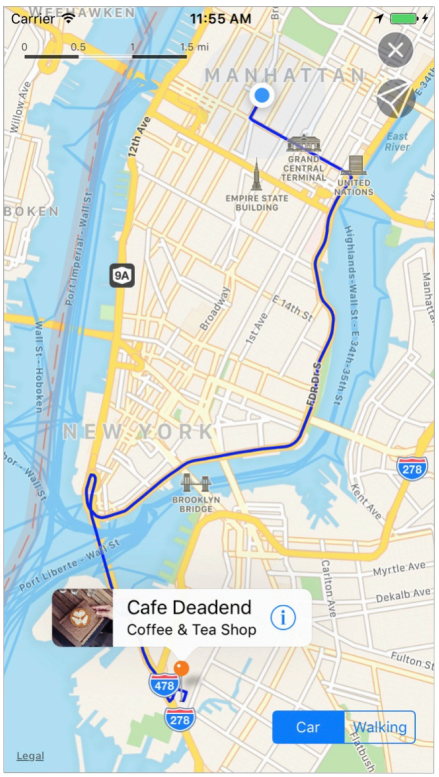
```
let rect = route.polyline.boundingMapRect

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

And place them right after the following line of code:

```
self.mapView.add(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.

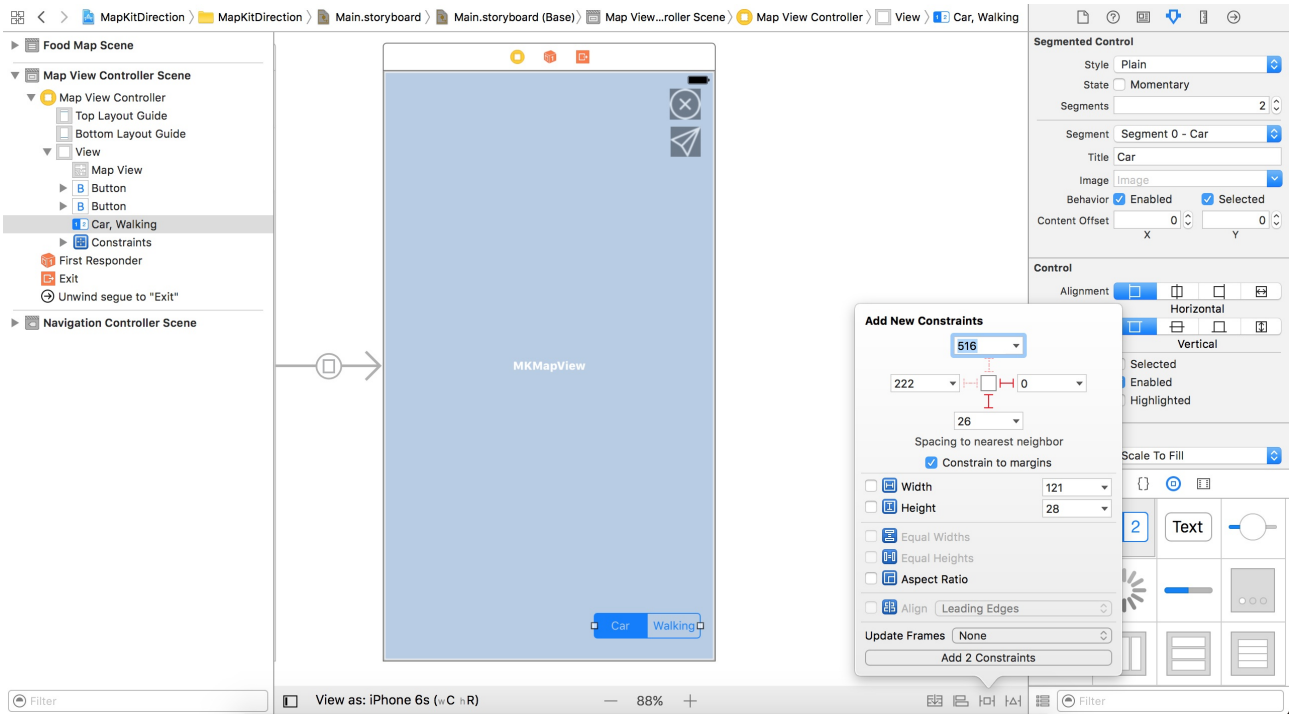


Using Segmented control

Presently, the app only provides route information for 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 to the navigation bar of the map view controller. Place it at the lower corner. 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, click the Pin button to add a couple of auto layout constraints. Your UI should look similar to figure 8.9.



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

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

Go back to the storyboard and connect the segmented control with the outlet variable. In the viewDidLoad method of MapViewController.swift , put this line of code right after super.viewDidLoad() :

```
segmentedControl.isHidden = true
```

We only want to display the control 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 introduction of this variable, we have to change the following line of code in the

showDirection method:

```
directionRequest.transportType = MKDirectionsTransportType.automobile
```

And replace MKDirectionsTransportType.automobile with currentTransportType like this: directionRequest.transportType = 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 a

ValueChanged event. So all you need to do is 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. But let's see how you can register the event by writing code.

Typically, you register the target-action methods for a segmented control like below. You can put the line of code after the initialization of segmentedControl :

```
segmentedControl.addTarget(self, action: #selector(showDirection), for:
.valueChanged)
```

Here, we 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).

The #selector syntax was first introduced in Swift 2.2. It can check the method you want to call to make sure it actually exists. In other words, if you do not have the showDirection method in your code, Xcode will warn you.

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

```
switch segmentedControl.selectedSegmentIndex {
case 0: currentTransportType = MKDirectionsTransportType.automobile
case 1: currentTransportType = MKDirectionsTransportType.walking
default: break
}
segmentedControl.isHidden = 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 in the calculate(completionHandler:) closure:

```
self.mapView.removeOverlays(self.mapView.overlays)
```

Place the line of code right before calling the add(_:level:) method. Your closure should look like this:

```
directions.calculate { (routeResponse, routeError) -> Void in
    guard let routeResponse = routeResponse else {
        if let routeError = routeError {
            print("Error: \(routeError)")
        }
        return

    }

    let route = routeResponse.routes[0]
    self.mapView.removeOverlays(self.mapView.overlays)
    self.mapView.add(route.polyline, level: MKOverlayLevel.aboveRoads)
    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.

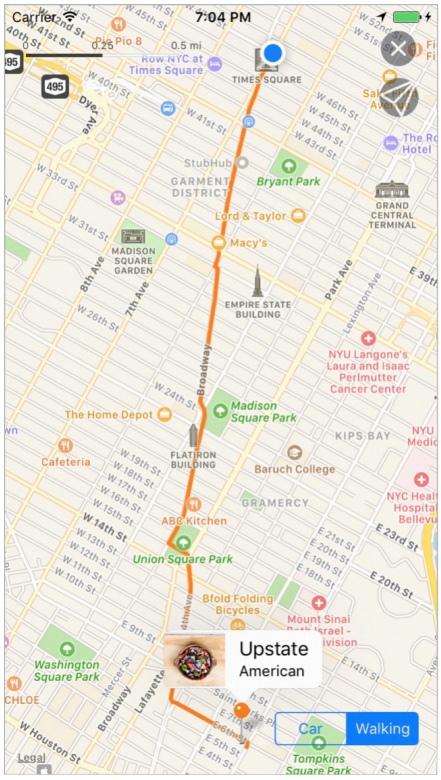
You can now test the app. In the map view, tap the Direction button and the segmented control should appear. You're free to select the Walking segment to display the walking directions.

For now, both types of routes are shown in blue. You can make a minor change in the mapView(_:rendererFor:) method of the MapViewController class to display a different color.

Simply change this line of code:

```
renderer.strokeColor = (currentTransportType == .automobile) ? UIColor.blue :
UIColor.orange
```

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 can 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. Each step in a route corresponds to a single instruction that would need to be followed by the user.

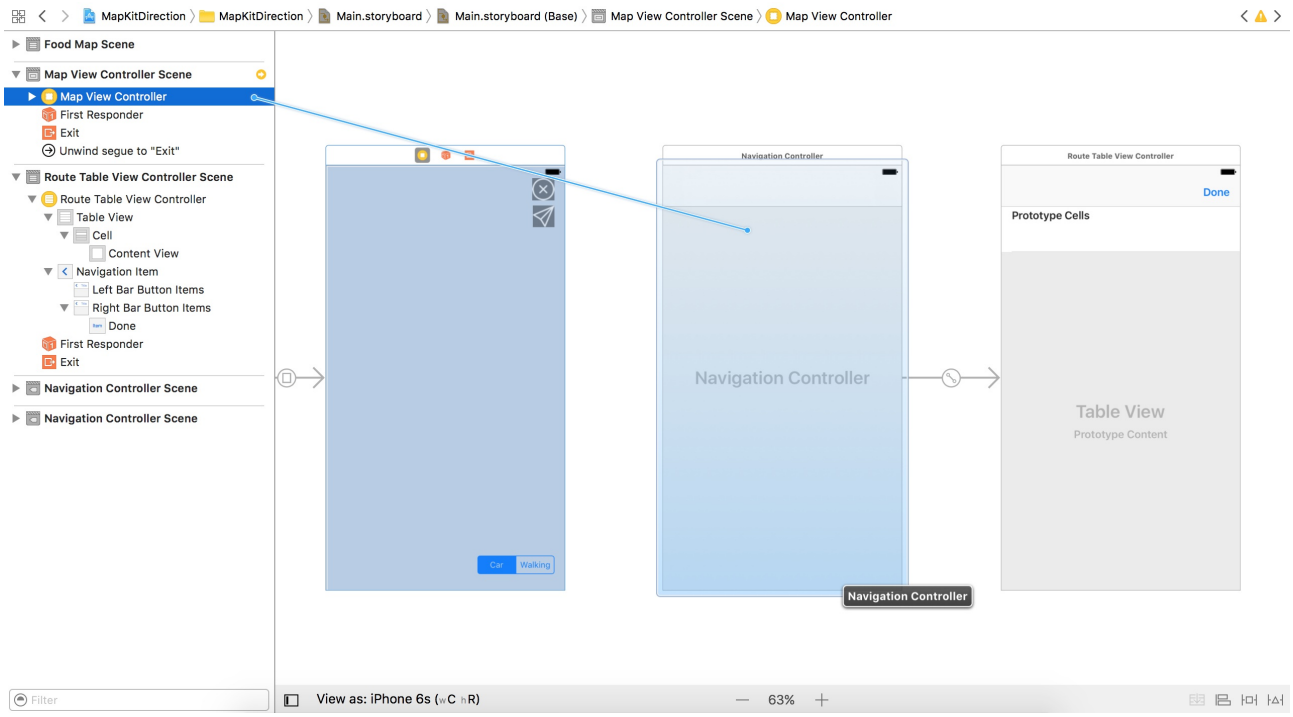
Okay, let's tweak the demo. When someone taps the annotation, the app will display the detailed driving/walking instructions.

First, add a table view controller to the storyboard and set the identifier of the prototype cell as to Cell . Next, embed the table view controller in a navigation controller, and change the title of the

navigation bar to "Steps". Also, add a bar button item to the navigation bar. In the Attributes inspector, change the system item option to Done .

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 present modally 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 .

You may have a couple of questions:

How can we get the detailed steps from the route?
How do we know if a user touches the annotation in a map?

As we 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, MKAnnotationView provides an optional accessory view displayed on the right side of a standard callout bubble. Once you create the accessory view, the following method of your map view's delegate will be called when a user taps the accessory view:

```
optional func mapView(_ mapView: MKMapView, annotationView view:
MKAnnotationView, calloutAccessoryControlTapped control: UIControl)
```

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 import MapKit:

```
import MapKit
```

Next, 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 numberOfSections(in tableView: UITableView) -> Int {
    // Return the number of sections
    return 1
}
```

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
    // Return the number of rows
    return routeSteps.count
}
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for:
                indexPath)
    // 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 touch of an annotation.

At the very beginning of the class, declare a new variable to store the current route: `var currentRoute: MKRoute?`

In the `mapView(_:viewFor:)` method, insert the following line of code before `return annotationView` :

```
annotationView?.rightCalloutAccessoryView = UIButton(type:
UIButtonType.detailDisclosure)
```

Here we add a detail disclosure button to the right side of an annotation. To handle a touch, we implement the `mapView(_:annotationView:calloutAccessoryControlTapped:)` method like this:

```
func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView,
calloutAccessoryControlTapped control: UIControl) {
    performSegue(withIdentifier: "showSteps", sender: view)
}
```

In iOS, you're allowed to trigger a segue programmatically by calling the performSegue(withIdentifier:sender:) 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 bring up the Steps table view controller when the above performSegue(withIdentifier:sender:) method is called.

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

In the body of the calculate(completionHandler:) closure, insert a line of code to update the current route:

```
self.currentRoute = route
```

It should be placed right before calling the removeOverlays method. The closure should look like this after the modification:

```
directions.calculate { (routeResponse, routeError) -> Void in
    guard let routeResponse = routeResponse else {
        if let routeError = routeError {
            print("Error: \(routeError)")
        }
        return

    }

    let route = routeResponse.routes[0]
    self.currentRoute = route
    self.mapView.removeOverlays(self.mapView.overlays)
    self.mapView.add(route.polyline, level: MKOverlayLevel.aboveRoads)
    let rect = route.polyline.boundingMapRect
    self.mapView.setRegion(MKCoordinateRegionForMapRect(rect), animated: true)
}
```

To pass the route steps to RouteTableViewController , implement the performSegue(withIdentifier:sender:) method like this:

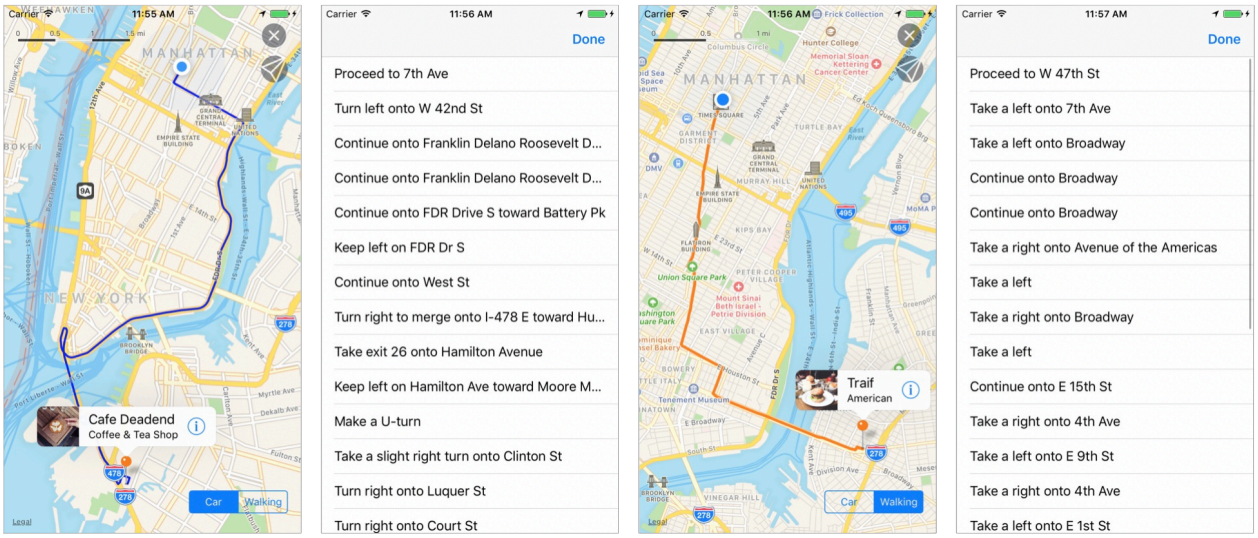
```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.
    if segue.identifier == "showSteps" {
        let routeTableViewController =
        segue.destination.childViewControllers[0] as! RouteTableViewController
        if let steps = currentRoute?.steps {
            routeTableViewController.routeSteps = steps
        }

    }

}
```

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

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

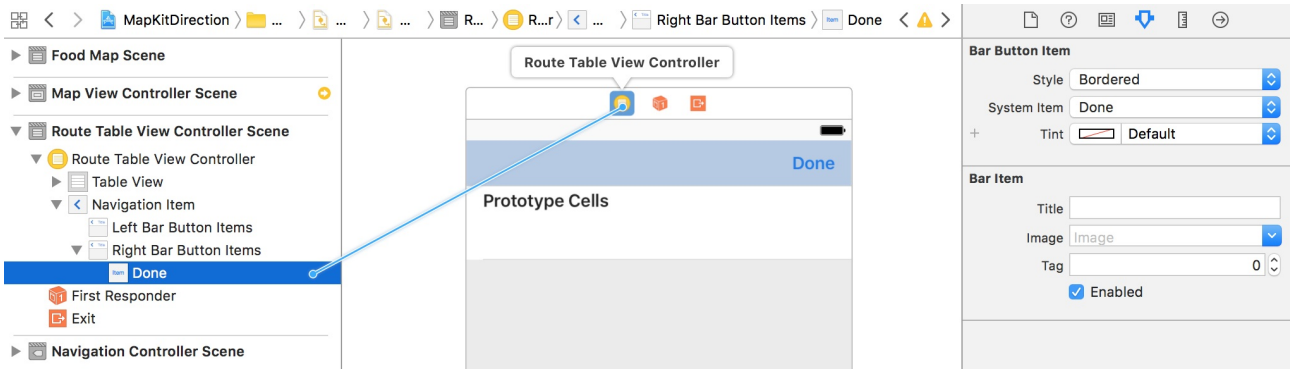


But we still miss one thing. When you tap the Done button in the route table view controller, it doesn't dismiss the controller. To make it work, create an action method in the

RouteTableViewController class:

```
@IBAction func close() {
    dismiss(animated: true, completion: nil)
}
```

Then connect the Done button with the close() method in the storyboard.



***