raywenderlich.com

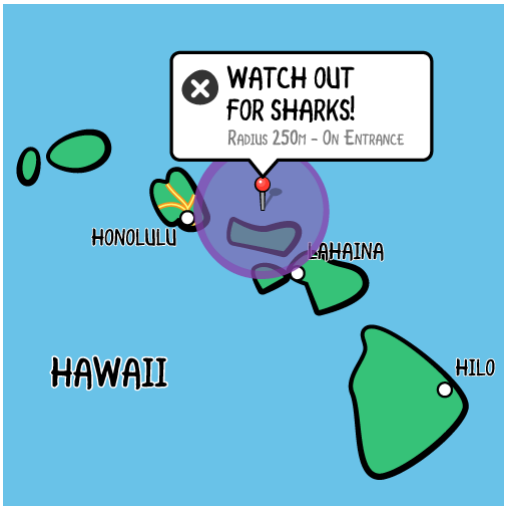
Geofencing Tutorial with Core Location

Andy Pereira on September 12, 2016

Update Note: This tutorial has been updated for Xcode 8 / Swift 3.

Geofencing notifies your app when its device enters or leaves geographical regions you set up. It lets you make cools that can trigger a notification whenever you leave home, or greet users with the latest and greatest deals whenever favorite shops are nearby. In this geofencing tutorial, you'll learn how to use region monitoring in iOS with Swift – using the Region Monitoring API in Core Location.

In particular, you'll create a location-based reminder app called Geotify that will let the user create reminders and associate them with real-world locations. Let's get started!



Let's get geofencing!

Getting Started

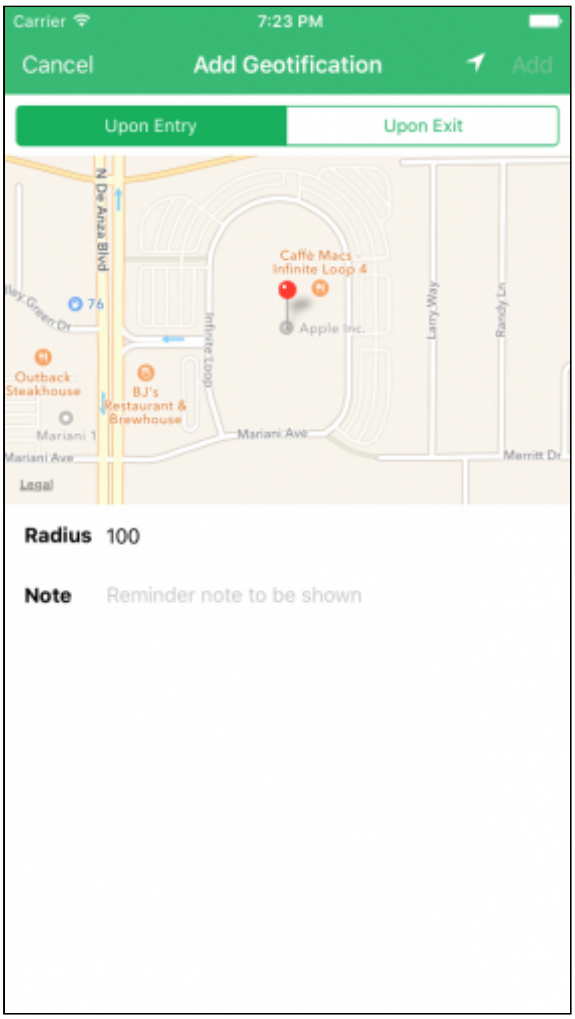
Download the starter project. The project provides a simple user interface for adding/removing annotation items to/from a map view. Each annotation item represents a reminder with a location, or as I like to call it, a geotification. :]

Build and run the project, and you'll see an empty map view.

Tap on the **+** button on the navigation bar to add a new geotification. The app will present a separate view, allowing you to set up various properties for your geotification.

For this tutorial, you will add a pin on Apple's headquarters in Cupertino. If you don't know where it is, open this google map in a separate tab and use it to hunt the right spot. Be sure to zoom in to make the pin nice and accurate!

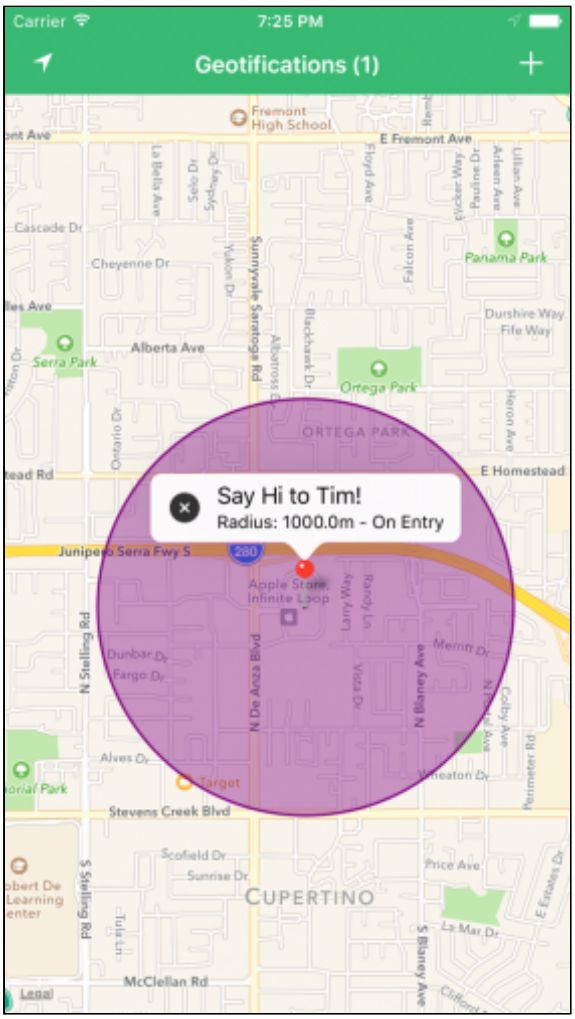
> **Note:** To pinch to zoom on the simulator, hold down option, then hold shift temporarily to move the pinch center, then release shift and click-drag to pinch.



The **radius** represents the distance in meters from the specified location, at which iOS will trigger the notification. The **note** can be any message you wish to display during the notification. The app also lets the user specify whether it should trigger the reminder upon either entry or exit of the defined circular geofence, via the segmented control at the top.

Enter **1000** for the radius value and **Say Hi to Tim!** for the note, and leave it as **Upon Entry** for your first geotification.

Click **Add** once you're satisfied with all the values. You'll see your geotification appear as a new annotation pin on the map view, with a circle around it denoting the defined geofence:



Tap on the pin and you'll reveal the geotification's details, such as the reminder note and the event type you specified earlier. Don't tap on the little cross unless you want to delete the geotification!

Feel free to add or remove as many geotifications as you want. As the app uses `NSUserDefaults` as a persistence store, the list of geotifications will persist between relaunches.

Setting Up a Location Manager and Permissions

At this point, any geotifications you've added to the map view are only for visualization. You'll fix this by taking each geotification and registering its associated geofence with Core Location for monitoring.

Before any geofence monitoring can happen, though, you need to set up a Location Manager instance and request the appropriate permissions.

Open **GeotificationsViewController.swift** and declare a constant instance of a `CLLocationManager` near the top of the class, as shown below:

```
class GeotificationsViewController: UIViewController {

  @IBOutlet weak var mapView: MKMapView!

  var geotifications = [Geotification]()
  let locationManager = CLLocationManager() // Add this statement

  ...
}
```

Next, replace `viewDidLoad()` with the following code:

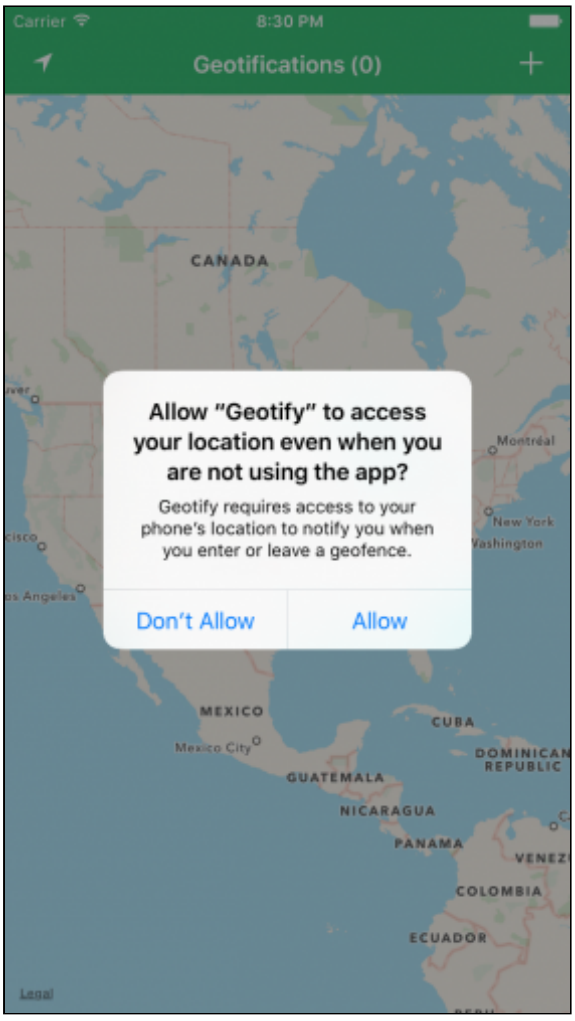
```
override func viewDidLoad() {
  super.viewDidLoad()
  // 1
  locationManager.delegate = self
  // 2
  locationManager.requestAlwaysAuthorization()
  // 3
  loadAllGeotifications()
}
```

Let's run through this method step by step:

1. You set the view controller as the delegate of the `locationManager` instance so that the view controller can receive the relevant delegate method calls.
2. You make a call to `requestAlwaysAuthorization()`, which invokes a prompt to the user requesting for `Always` authorization to use location services. Apps with geofencing capabilities need `Always` authorization, due to the need to monitor geofences even when the app isn't running. **Info.plist** has already been setup with a message to show the user when requesting the user's location under the key `NSLocationAlwaysUsageDescription`.
3. You call `loadAllGeotifications()`, which deserializes the list of geotifications previously saved to `NSUserDefaults` and loads them into a local geotifications array. The method also loads the geotifications as annotations on the map view.

When the app prompts the user for authorization, it will show `NSLocationAlwaysUsageDescription`, a user-friendly explanation of why the app requires access to the user's location. This key is mandatory when you request authorization for location services. If it's missing, the system will ignore the request and prevent location services from starting altogether.

Build and run the project, and you'll see a user prompt with the aforementioned description that's been set:



You've set up your app to request the required permission. Great! Click or tap Allow to ensure the location manager will receive delegate callbacks at the appropriate times.

Before you proceed to implement the geofencing, there's a small issue you have to resolve: the user's current location isn't showing up on the map view! This feature is disabled, and as a result, the **zoom** button on the top-left of the navigation bar doesn't work.

Fortunately, the fix is not difficult — you'll simply enable the current location only after the app is authorized.

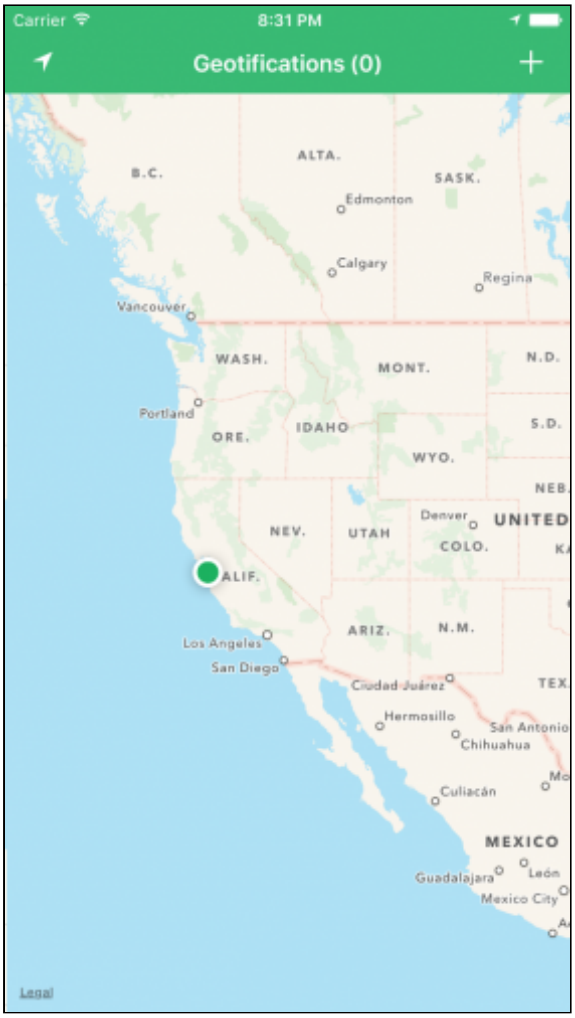
In **GeotificationsViewController.swift**, add the following delegate method to the `CLLocationManagerDelegate` extension:

```
extension GeotificationsViewController: CLLocationManagerDelegate {
  func locationManager(_ manager: CLLocationManager, didChangeAuthorization status:
CLAuthorizationStatus) {
    mapView.showsUserLocation = (status == .authorizedAlways)
  }
}
```

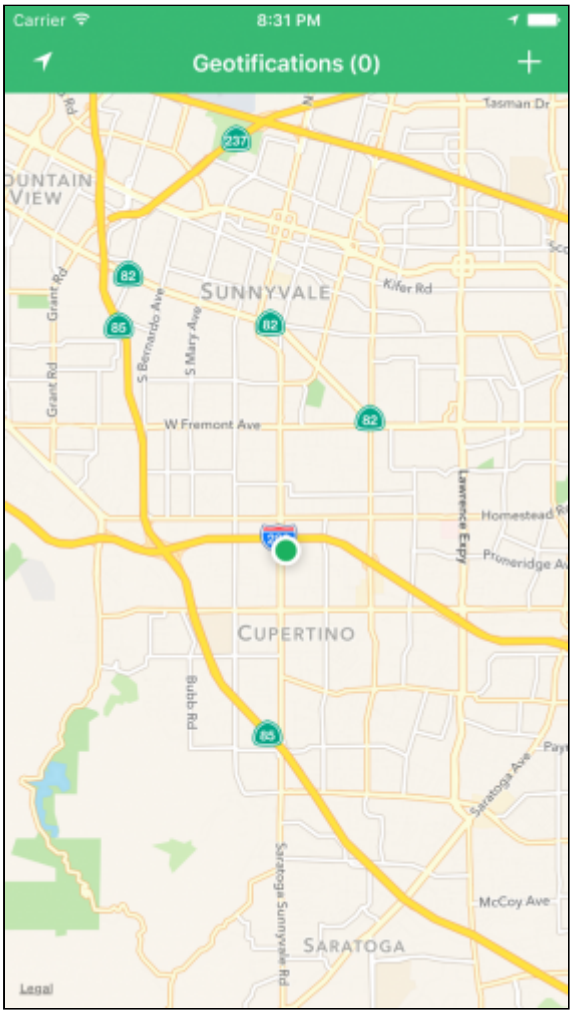
The location manager calls `locationManager(_:didChangeAuthorizationStatus:)` whenever the authorization status changes. If the user has already granted the app permission to use Location Services, this method will be called by the location manager after you've initialized the location manager and set its delegate.

That makes this method an ideal place to check if the app is authorized. If it is, you enable the map view to show the user's current location.

Build and run the app. If you're running it on a device, you'll see the location marker appear on the main map view. If you're running on the simulator, click **Debug\Location\Apple** in the menu to see the location marker:



In addition, the **zoom** button on the navigation bar now works. :]



Registering Your Geofences

With the location manager properly configured, the next order of business is to allow your app to register user geofences for monitoring.

In your app, the user geofence information is stored within your custom `Geotification` model. However, Core Location requires each geofence to be represented as a `CLCircularRegion` instance before it can be registered for monitoring. To handle this requirement, you'll create a helper method that returns a `CLCircularRegion` from a given `Geotification` object.

Open **GeotificationsViewController.swift** and add the following method to the main body:

```
func region(withGeotification geotification: Geotification) -> CLCircularRegion {
  // 1
  let region = CLCircularRegion(center: geotification.coordinate, radius:
geotification.radius, identifier: geotification.identifier)
  // 2
  region.notifyOnEntry = (geotification.eventType == .onEntry)
  region.notifyOnExit = !region.notifyOnEntry
  return region
}
```

Here's what the above method does:

1. You initialize a `CLCircularRegion` with the location of the geofence, the radius of the geofence and an identifier that allows iOS to distinguish between the registered geofences of a given app. The initialization is rather straightforward, as the `Geotification` model already contains the required properties.

2. The `CLCircularRegion` instance also has two Boolean properties, `notifyOnEntry` and `notifyOnExit`. These flags specify whether geofence events will be triggered when the device enters and leaves the defined geofence, respectively. Since you're designing your app to allow only one notification type per geofence, you set one of the flags to `true` while you set the other to `false`, based on the `enum` value stored in the `Geotification` object.

Next, you need a method to start monitoring a given geotification whenever the user adds one.

Add the following method to the body of `GeotificationsViewController`:

```
func startMonitoring(geotification: Geotification) {
  // 1
  if !CLLocationManager.isMonitoringAvailable(for: CLCircularRegion.self) {
    showAlert(withTitle:"Error", message: "Geofencing is not supported on this device!")
    return
  }
  // 2
  if CLLocationManager.authorizationStatus() != .authorizedAlways {
    showAlert(withTitle:"Warning", message: "Your geotification is saved but will only be
activated once you grant Geotify permission to access the device location.")
  }
  // 3
  let region = self.region(withGeotification: geotification)
  // 4
  locationManager.startMonitoring(for: region)
}
```

Let's walk through the method step by step:

1. `isMonitoringAvailableForClass(_:)` determines if the device has the required hardware to support the monitoring of geofences. If monitoring is unavailable, you bail out entirely and alert the user accordingly. `showSimpleAlertWithTitle(_:message:viewController)` is a helper function in **Utilities.swift** that takes in a title and message and displays an alert view.

2. Next, you check the authorization status to ensure that the app has also been granted the required permission to use Location Services. If the app isn't authorized, it won't receive any geofence-related notifications.
However, in this case, you'll still allow the user to save the geotification, since Core Location lets you register geofences even when the app isn't authorized. When the user subsequently grants authorization to the app, monitoring for those geofences will begin automatically.

3. You create a `CLCircularRegion` instance from the given geotification using the helper method you defined earlier.

4. Finally, you register the `CLCircularRegion` instance with Core Location for monitoring.

With your start method done, you also need a method to *stop* monitoring a given geotification when the user removes it from the app.

In **GeotificationsViewController.swift**, add the following method below `startMonitoringGeotificiation(_:)`:

```
func stopMonitoring(geotification: Geotification) {
  for region in locationManager.monitoredRegions {
    guard let circularRegion = region as? CLCircularRegion, circularRegion.identifier ==
geotification.identifier else { continue }
    locationManager.stopMonitoring(for: circularRegion)
  }
}
```

The method simply instructs the `locationManager` to stop monitoring the `CLCircularRegion` associated with the given geotification.

Now that you have both the start and stop methods complete, you'll use them whenever you add or remove a geotification. You'll begin with the adding part.

First, take a look at `addGeotificationViewController(_:didAddCoordinate)` in **GeotificationsViewController.swift**.

The method is the delegate call invoked by the `AddGeotificationViewController` upon creating a geotification; it's responsible for creating a new `Geotification` object using the values passed from `AddGeotificationsViewController`, and updating both the map view and the `geotifications` list accordingly. Then you call `saveAllGeotifications()`, which takes the newly-updated `geotifications` list and persists it via `NSUserDefaults`.

Now, replace the method with the following code:

```
func addGeotificationViewController(controller: AddGeotificationViewController,
didAddCoordinate coordinate: CLLocationCoordinate2D, radius: Double, identifier: String,
note: String, eventType: EventType) {
  controller.dismiss(animated: true, completion: nil)
  // 1
  let clampedRadius = min(radius, locationManager.maximumRegionMonitoringDistance)
  let geotification = Geotification(coordinate: coordinate, radius: clampedRadius,
identifier: identifier, note: note, eventType: eventType)
  add(geotification: geotification)
  // 2
  startMonitoring(geotification: geotification)
  saveAllGeotifications()
}
```

You've made two key changes to the code:

1. You ensure that the value of the radius is clamped to the `maximumRegionMonitoringDistance` property of `locationManager`, which is defined as the largest radius in meters that can be assigned to a geofence. This is important, as any value that exceeds this maximum will cause monitoring to fail.
2. You add a call to `startMonitoringGeotification(_:)` to ensure that the geofence associated with the newly-added geotification is registered with Core Location for monitoring.

At this point, the app is fully capable of registering new geofences for monitoring. There is, however, a limitation: As geofences are a shared system resource, Core Location restricts the number of registered geofences to a maximum of 20 per app.

While there are workarounds to this limitation (See **Where to Go From Here?** for a short discussion), for the purposes of this tutorial, you'll take the approach of limiting the number of geotifications the user can add.

Add a line to `updateGeotificationsCount()`, as shown in the code below:

```
func updateGeotificationsCount() {
  title = "Geotifications (\(geotifications.count))"
  navigationItem.rightBarButtonItem?.isEnabled = (geotifications.count < 20)  // Add this
line
}
```

This line disables the **Add** button in the navigation bar whenever the app reaches the limit.

Finally, let's deal with the removal of geotifications. This functionality is handled in `mapView(_:annotationView:calloutAccessoryControlTapped:)`, which is invoked whenever the user taps the "delete" accessory control on each annotation.

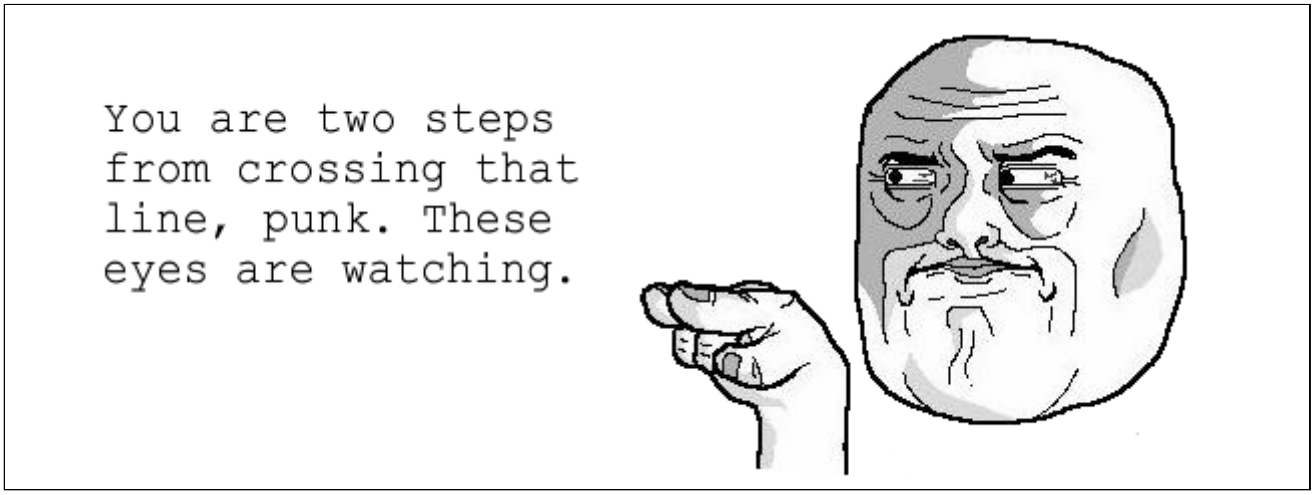
Add a call to `stopMonitoring(geotification:)` to `mapView(_:annotationView:calloutAccessoryControlTapped:)`, as shown below:

```
func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView,
calloutAccessoryControlTapped control: UIControl) {
  // Delete geotification
  let geotification = view.annotation as! Geotification
  stopMonitoring(geotification: geotification)   // Add this statement
  removeGeotification(geotification)
  saveAllGeotifications()
}
```

The additional statement stops monitoring the geofence associated with the geotification, before removing it and saving the changes to **NSUserDefaults**.

At this point, your app is fully capable of monitoring and un-monitoring user geofences. Hurray!

Build and run the project. You won't see any changes, but the app will now be able to register geofence regions for monitoring. However, it won't be able to react to any geofence events just yet. Not to worry—that will be your next order of business!



Reacting to Geofence Events

You'll start by implementing some of the delegate methods to facilitate error handling – these are important to add in case anything goes wrong.

In **GeotificationsViewController.swift**, add the following methods to the **CLLocationManagerDelegate** extension:

```
func locationManager(_ manager: CLLocationManager, monitoringDidFailFor region: CLRegion?,
withError error: Error) {
  print("Monitoring failed for region with identifier: \(region!.identifier)")
}

func locationManager(_ manager: CLLocationManager, didFailWithError error: Error) {
  print("Location Manager failed with the following error: \(error)")
}
```

These delegate methods simply log any errors that the location manager encounters to facilitate your debugging.

> **Note:** You'll definitely want to handle these errors more robustly in your production apps. For example, instead of failing silently, you could inform the user what went wrong.

Next, open **AppDelegate.swift**; this is where you'll add code to properly listen and react to geofence entry and exit events.

Add the following line at the top of the file to import the **CoreLocation** framework:

```
import CoreLocation
```

Ensure that the **AppDelegate** has a **CLLocationManager** instance near the top of the class, as shown below:

```
class AppDelegate: UIResponder, UIApplicationDelegate {
  var window: UIWindow?

  let locationManager = CLLocationManager() // Add this statement
  ...
}
```

Replace `application(_:didFinishLaunchingWithOptions:)` with the following implementation:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey : Any]? = nil) -> Bool {
  locationManager.delegate = self
  locationManager.requestAlwaysAuthorization()
  return true
}
```

You've set up your `AppDelegate` to receive geofence-related events. But you might wonder, "Why did I designate the `AppDelegate` to do this instead of the view controller?"

Geofences registered by an app are monitored at all times, including when the app isn't running. If the device triggers a geofence event while the app isn't running, iOS automatically relaunches the app directly into the background. This makes the `AppDelegate` an ideal entry point to handle the event, as the view controller may not be loaded or ready.

Now you might also wonder, "How will a newly-created `CLLocationManager` instance be able to know about the monitored geofences?"

It turns out that all geofences registered by your app for monitoring are conveniently accessible by all location managers in your app, so it doesn't matter where the location managers are initialized. Pretty nifty, right? :]

Now all that's left is to implement the relevant delegate methods to react to the geofence events. Before you do so, you'll create a method to handle a geofence event.

Add the following method to **AppDelegate.swift**:

```
func handleEvent(forRegion region: CLRegion!) {
  print("Geofence triggered!")
}
```

At this point, the method takes in a `CLRegion` and simply logs a statement. Not to worry—you'll implement the event handling later.

Next, add the following delegate methods in the `CLLocationManagerDelegate` extension of **AppDelegate.swift**, as well as a call to the `handleRegionEvent(_:)` function you just created, as shown in the code below:

```
extension AppDelegate: CLLocationManagerDelegate {

  func locationManager(_ manager: CLLocationManager, didEnterRegion region: CLRegion) {
    if region is CLCircularRegion {
      handleEvent(forRegion: region)
    }
  }

  func locationManager(_ manager: CLLocationManager, didExitRegion region: CLRegion) {
    if region is CLCircularRegion {
      handleEvent(forRegion: region)
    }
  }
}
```

As the method names aptly suggest, you fire `locationManager(_:didEnterRegion:)` when the device enters a `CLRegion`, while you fire `locationManager(_:didExitRegion:)` when the device exits a `CLRegion`.

Both methods return the `CLRegion` in question, which you need to check to ensure it's a `CLCircularRegion`, since it could be a `CLBeaconRegion` if your app happens to be monitoring iBeacons, too. If the region is indeed a `CLCircularRegion`, you accordingly call `handleRegionEvent(_:)`.

> **Note:** A geofence event is triggered only when iOS detects a boundary crossing. If the user is already within a geofence at the point of registration, iOS won't generate an event. If you need to query whether the device location falls within or outside a given geofence, Apple provides a method called `requestStateForRegion(_:)`.

Now that your app is able to receive geofence events, you're ready to give it a proper test run. If that doesn't excite you, it really ought to, because for the first time in this tutorial, you're going to see some results. :]

The most accurate way to test your app is to deploy it on your device, add some geotifications and take the app for a walk or a drive. However, it wouldn't be wise to do so right now, as you wouldn't be able to verify the print logs emitted by the geofence events with the device unplugged. Besides, it would be nice to get assurance that the app works before you commit to taking it for a spin.

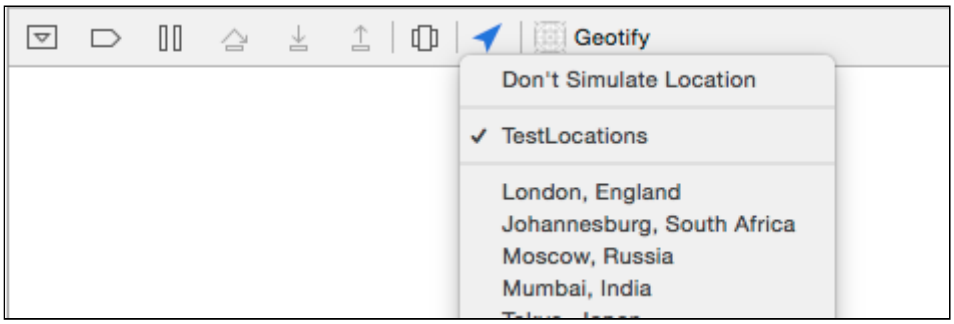
Fortunately, there's an easy way do this without leaving the comfort of your home. Xcode lets you include a hardcoded waypoint GPX file in your project that you can use to simulate test locations. Lucky for you, the starter project includes one for your convenience. :]

Open up **TestLocations.gpx**, which you can find in the **Supporting Files** group, and inspect its contents. You'll see the following:

```
<?xml version="1.0"?>
<gpx version="1.1" creator="Xcode">
  <wpt lat="37.422" lon="-122.084058">
    <name>Google</name>
  </wpt>
  <wpt lat="37.3270145" lon="-122.0310273">
    <name>Apple</name>
  </wpt>
</gpx>
```

The GPX file is essentially an XML file that contains two waypoints: **Google's Googleplex** in Mountain View and **Apple's Headquarters** in Cupertino.

To begin simulating the locations in the GPX file, build and run the project. When the app launches the main view controller, go back to **Xcode**, select the **Location** icon in the **Debug bar** and choose **TestLocations**:

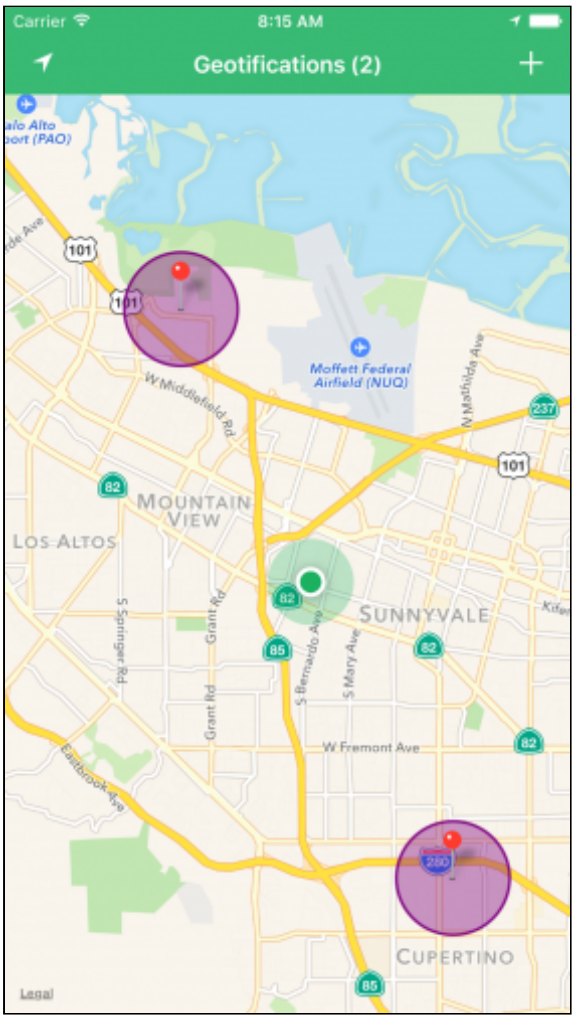


Back in the app, use the **Zoom** button on the top-left of the navigation bar to zoom to the current location. Once you get close to the area, you'll see the location marker moving repeatedly from the Googleplex to Apple, Inc. and back.

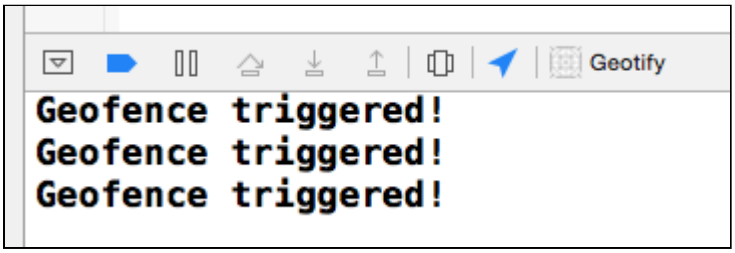
Test the app by adding a few geotifications along the path defined by the two waypoints. If you added any geotifications earlier in the tutorial before you enabled geofence registration, those geotifications will obviously not work, so you might want to clear them out and start afresh.

For the test locations, it's a good idea to place a geotification roughly at each waypoint. Here's a possible test scenario:

- **Google:** Radius: 1000m, Message: "Say Bye to Google!", Notify on Exit
- **Apple:** Radius: 1000m, Message: "Say Hi to Apple!", Notify on Entry



Once you've added your geotifications, you'll see a log in the console each time the location marker enters or leaves a geofence. If you activate the home button or lock the screen to send the app to the background, you'll also see the logs each time the device crosses a geofence, though you obviously won't be able to verify that behavior visually.



Note: Location simulation works both in iOS Simulator and on a real device. However, the iOS Simulator can be quite inaccurate in this case; the timings of the triggered events do not coincide very well with the visual movement of the simulated location in and out of each geofence. You would do better to simulate locations on your device, or better still, take the app for a walk!

Notifying the User of Geofence Events

You've made a lot of progress with the app. At this point, it simply remains for you to notify the user whenever the device crosses the geofence of a geotification—so prepare yourself to do just that.

To obtain the note associated with a triggering **CLCircularRegion** returned by the delegate calls, you need to retrieve the corresponding geotification that was persisted in **NSUserDefaults**. This turns out to be trivial, as you can use the unique identifier you assigned to the **CLCircularRegion** during registration to find the right geotification.

In **AppDelegate.swift**, add the following helper method at the bottom of the class:

```
func note(fromRegionIdentifier identifier: String) -> String? {
  let savedItems = UserDefaults.standard.array(forKey: PreferencesKeys.savedItems) as?
[NSData]
  let geotifications = savedItems?.map { NSKeyedUnarchiver.unarchiveObject(with: $0 as Data)
as? Geotification }
  let index = geotifications?.index { $0?.identifier == identifier }
  return index != nil ? geotifications?[index!]?.note : nil
}
```

This helper method retrieves the geotification note from the persistent store, based on its identifier, and returns the note for that geotification.

Now that you're able to retrieve the note associated with a geofence, you'll write code to trigger a notification whenever a geofence event is fired, using the note as the message.

Add the following statements to the end of **application(_:didFinishLaunchingWithOptions:)**, just before the method returns:

```
application.registerUserNotificationSettings(UIUserNotificationSettings(types: [.sound,
.alert, .badge], categories: nil))
UIApplication.shared.cancelAllLocalNotifications()
```

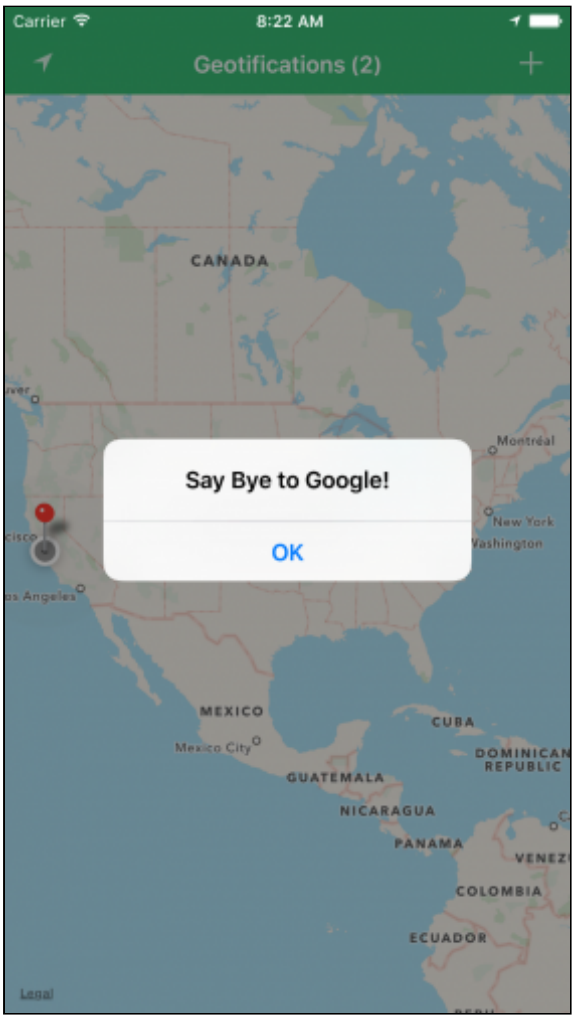
The code you've added prompts the user for permission to enable notifications for this app. In addition, it does some housekeeping by clearing out all existing notifications.

Next, replace the contents of **handleRegionEvent(_:)** with the following:

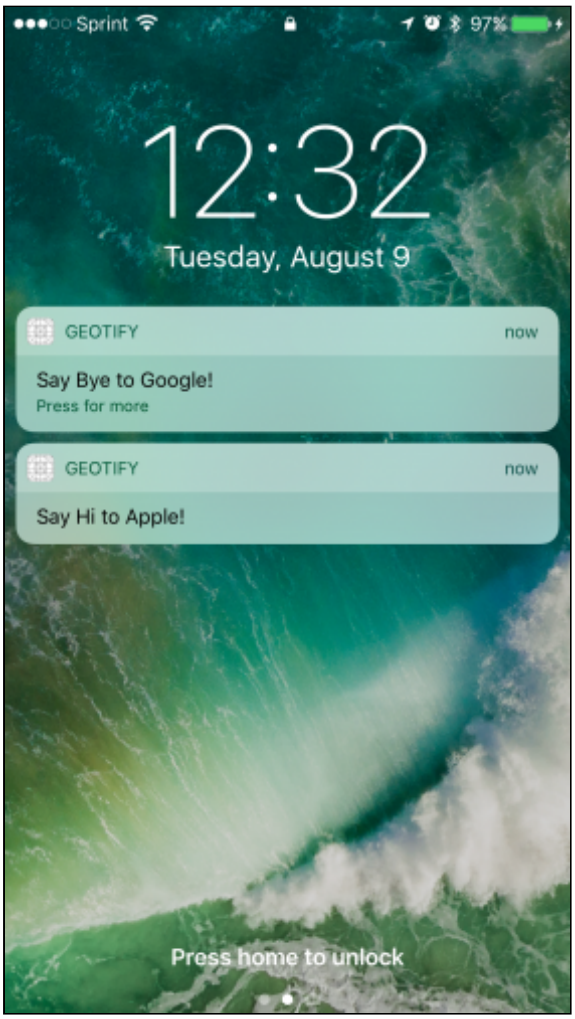
```
func handleEvent(forRegion region: CLRegion!) {
  // Show an alert if application is active
  if UIApplication.shared.applicationState == .active {
    guard let message = note(fromRegionIdentifier: region.identifier) else { return }
    window?.rootViewController?.showAlert(withTitle: nil, message: message)
  } else {
    // Otherwise present a local notification
    let notification = UILocalNotification()
    notification.alertBody = note(fromRegionIdentifier: region.identifier)
    notification.soundName = "Default"
    UIApplication.shared.presentLocalNotificationNow(notification)
  }
}
```

If the app is active, the code above simply shows an alert controller with the note as the message. Otherwise, it presents a location notification with the same message.

Build and run the project, and run through the test procedure covered in the previous section. Whenever your test triggers a geofence event, you'll see an alert controller displaying the reminder note:



Send the app to the background by activating the Home button or locking the device while the test is running. You'll continue to receive notifications periodically that signal geofence events:



And with that, you have a fully functional, location-based reminder app in your hands. And yes, get out there and take that app for a spin!

> **Note:** When you test the app, you may encounter situations where the notifications don't fire exactly at the point of boundary crossing.
>
> This is because before iOS considers a boundary as crossed, there is an additional cushion distance that must be traversed and a minimum time period that the device must linger at the new location. iOS internally defines these thresholds, seemingly to mitigate the spurious firing of notifications in the event the user is traveling very close to a geofence boundary.
>
> In addition, these thresholds seem to be affected by the available location hardware capabilities. From experience, the geofencing behavior is a lot more accurate when Wi-Fi is enabled on the device.

Where to Go From Here?

You can download the complete project for this tutorial here.

Congratulations—you're now equipped with the basic knowledge you need to build your own geofencing-enabled apps!

Geofencing is a powerful technology with many practical and far-reaching applications in such realms as marketing, resource management, security, parental control and even gaming—what you can achieve is really up to your imagination.

If you're looking for additional challenges or learning opportunities, consider the following:

- Allow the user to edit an existing geotification.

- Enable the app to handle more than 20 geofences by selectively monitoring geofences that are in the user's vicinity. One way to achieve this is by monitoring for significant location changes and selecting the 20 geofences closest to the user's current location.

I hope you've enjoyed this tutorial, and I really look forward to seeing how you use geofencing in your apps. Feel free to leave a comment or question below!

Andy Pereira

Andy is a software developer in Atlanta, GA. He is the author of a few personal iOS apps, as well as several B2B mobile apps.

You can find Andy on LinkedIn or Twitter.