物聯網感測元件實務

iBeacon Tutorial with iOS and Swift

王昱景　Brian Wang

brian.wang.frontline@gmail.com

- Have you ever wished that your phone could show your location inside a large building like a shopping mall or baseball stadium?

- Sure, GPS can give you an idea of which side of the building you are in.

- But good luck getting an accurate GPS signal in one of those steel and concrete sarcophaguses.

- What you need is something inside of the building to let your device determine its physical location.

- Enter iBeacons! In this iBeacons tutorial you'll create an app that lets you register known iBeacon emitters and tells you when your phone has moved outside of their range.

- The use case for this app is attaching an iBeacon emitter to your laptop bag, purse, or even your cat's collar — anything important you don't want to lose.

- Once your device moves outside the range of the emitter, your app detects this and notifies you.

- To continue with this tutorial, you'll need to test on a real iOS device and an iBeacon.

- If you don't have an iBeacon but have a second iOS device, you might be able to use it as a beacon; read on!

System Requirements

- iOS 10

- Swift 3

- Xcode 8.2.1

Getting Started

- There are many iBeacon devices available; a quick Google search should help reveal them to you.

- But when Apple introduced iBeacon, they also announced that any compatible iOS device could act as an iBeacon.

- The list currently includes the following devices:

  - iPhone 4s or later

  - 3rd generation iPad or later

  - iPad Mini or later

  - 5th generation iPod touch or later

- An iBeacon is nothing more than a Bluetooth Low Energy device that advertises information in a specific structure.

- Those specifics are beyond the scope of this tutorial, but the important thing to understand is that iOS can monitor for iBeacons that emit three values known as: **UUID**, **major** and **minor**.

- UUID is an acronym for universally unique identifier, which is a 128-bit value that's usually shown as a hex string like this: **B558CBDA-4472-4211-A350-FF1196FFE8C8**.

- In the context of iBeacons, a UUID is generally used to represent your top-level identity.

- Major and minor values provide a little more granularity on top of the UUID.

- These values are simply 16 bit unsigned integers that identify each individual iBeacon, even ones with the same UUID.

- For instance, if you owned multiple department stores you might have all of your iBeacons emit the same UUID, but each store would have its own major value, and each department within that store would have its own minor value.

- Your app could then respond to an iBeacon located in the shoe department of your Miami, Florida store.

ForgetMeNot Starter Project

- Download the starter project here — it contains a simple interface for adding and removing items from a table view.

- Each item in the table view represents a single iBeacon emitter, which in the real world translates to an item that you don't want to leave behind.

- Build and run the app; you'll see an empty list, devoid of items.

- Press the + button at the top right to add a new item as shown in the screenshot below:
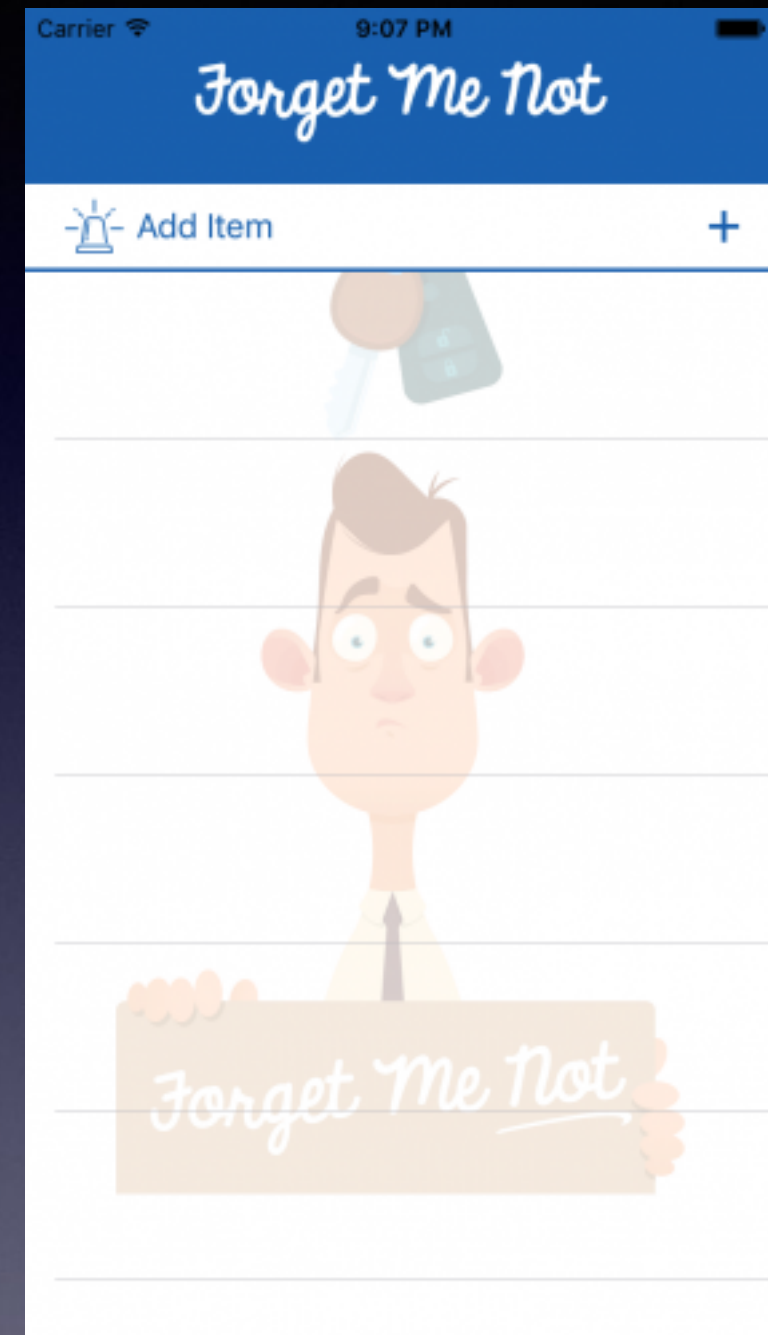
- To add an item, you simply enter a name for the item and the values corresponding to its iBeacon.

- You can find your iBeacon's UUID by reviewing your iBeacon's documentation – try adding it now, or use some placeholder values, as shown below:
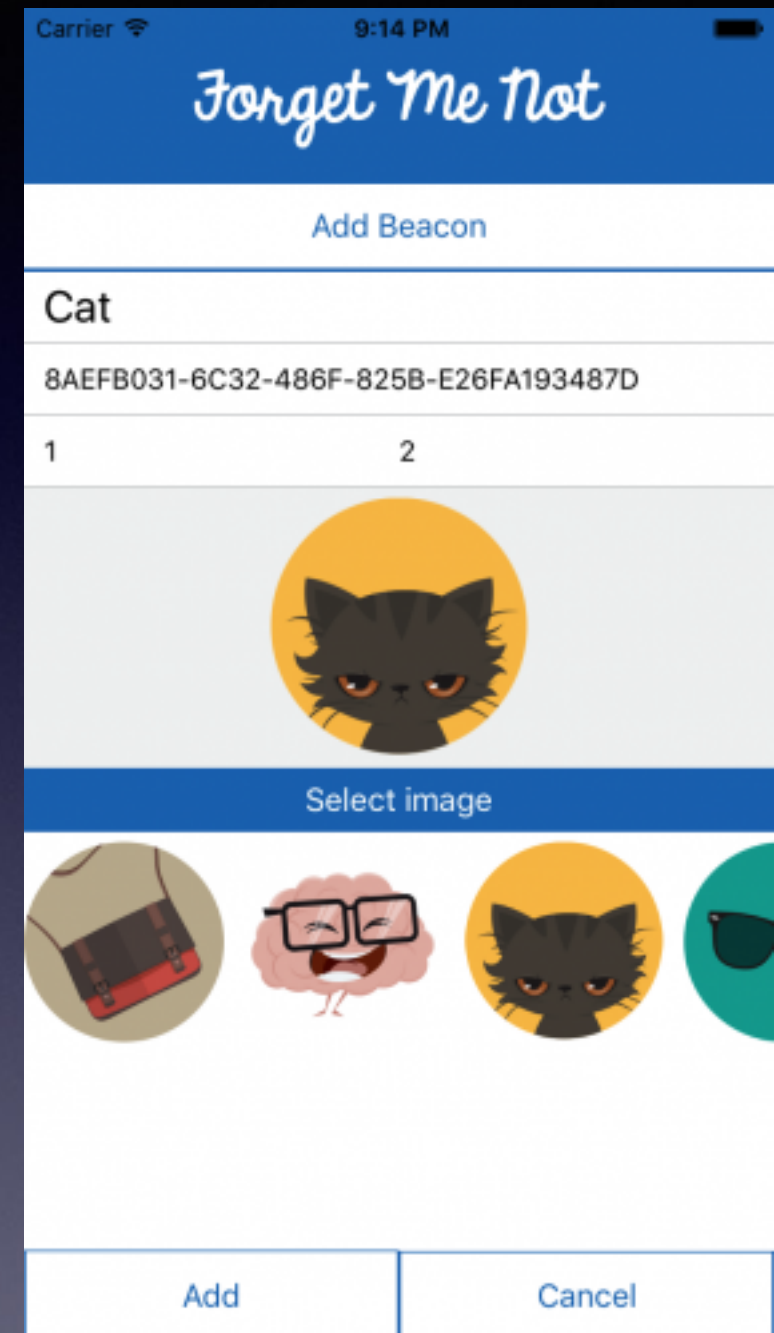
- Press **Save** to return to the list of items; you'll see your item with a location of Unknown, as shown below:
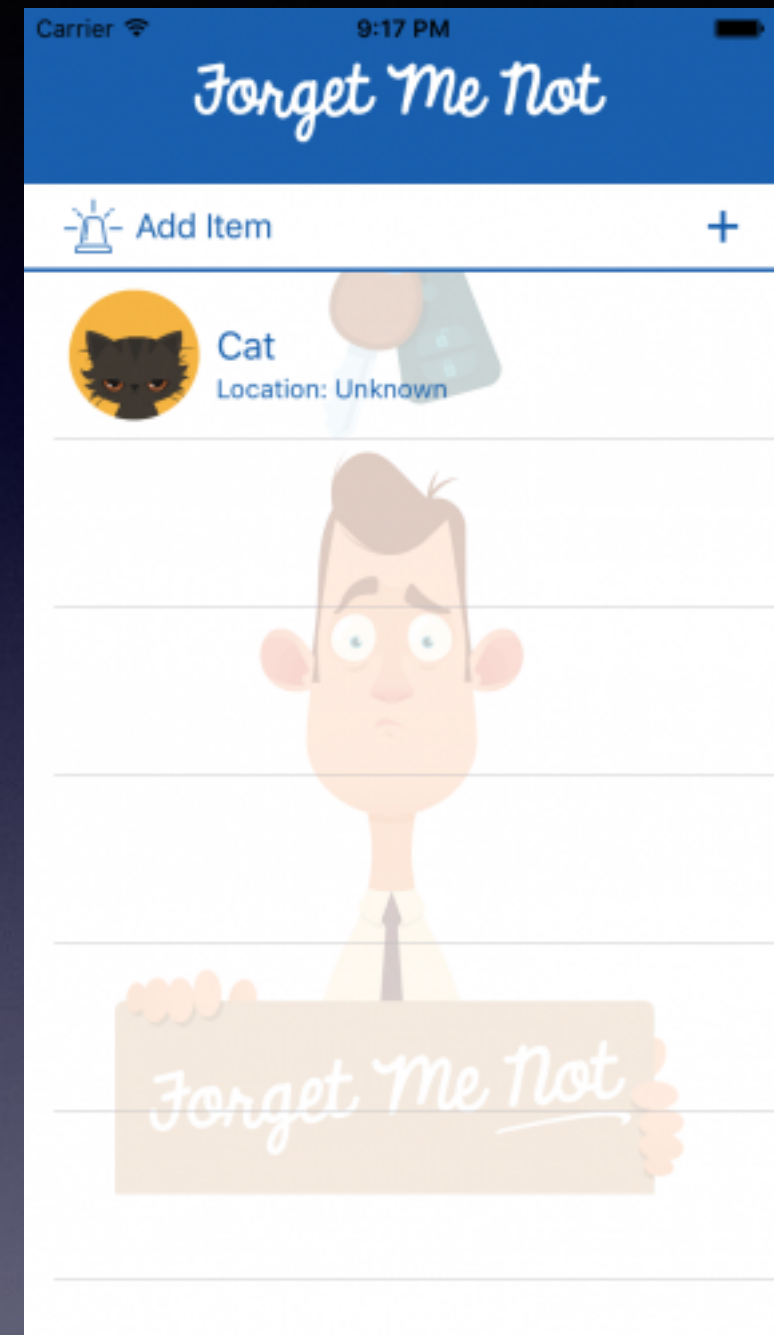
- You can add more items if you wish, or swipe to delete existing ones.

- **UserDefaults** persists the items in the list so that they're available when the user re-launches the app.

- On the surface it appears there's not much going on; most of the fun stuff is under the hood.

- The unique aspect in this app is the **Item** model class which represents the items in the list.

- Open **Item.swift** and have a look at it in Xcode.

- The model class mirrors what the interface requests from the user, and it conforms to **NSCoding** so that it can be serialized and deserialized to disk for persistence.

- Now take a look at **AddItemViewController.swift**.

- This is the controller for adding a new item. It's a simple **UIViewController**, except that it does some validation on user input to ensure that the user enters valid names and UUIDs.

- The **Add** button at the bottom left becomes tappable as soon as **txtName** and **txtUUID** are both valid.

- Now that you're acquainted with the starter project, you can move on to implementing the iBeacon bits into your project!

Core Location Permissions

- Your device won't listen for your iBeacon automatically — you have to tell it to do this first.

- The **CLBeaconRegion** class represents an iBeacon; the **CL** class prefix indicates that it is part of the Core Location framework.

- It may seem strange for an iBeacon to be related to Core Location since it's a Bluetooth device, but consider that iBeacons provide micro-location awareness while GPS provides macro-location awareness.

- You would leverage the Core Bluetooth framework for iBeacons when programming an iOS device to act as an iBeacon, but when monitoring for iBeacons you only need to work with Core Location.

- Your first order of business is to adapt the **Item** model for **CLBeaconRegion**.

- Open **Item.swift** and add the following import to the top of the file:

```
import CoreLocation
```

- Next, change the **majorValue** and **minorValue** definitions as well as the initializer as follows:

```swift
let majorValue: CLBeaconMajorValue
let minorValue: CLBeaconMinorValue

init(name: String, icon: Int, uuid: UUID, majorValue: Int, minorValue: Int) {
  self.name = name
  self.icon = icon
  self.uuid = uuid
  self.majorValue = CLBeaconMajorValue(majorValue)
  self.minorValue = CLBeaconMinorValue(minorValue)
}
```

- **CLBeaconMajorValue** and **CLBeaconMinorValue** are both a **typealias** for **UInt16**, and are used for representing major and minor values in the **CoreLocation** framework.

- Although the underlying data type is the same, this improves readability of the model and adds type safety so you don't mix up major and minor values.

- Open **ItemsViewController.swift**, add the Core Location import to the top of the file:

```
import CoreLocation
```

- Add the following property to the **ItemsViewController** class:

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

- You'll use this **CLLocationManager** instance as your entry point into Core Location.

- Next, replace **viewDidLoad()** with the following:

```
override func viewDidLoad() {
  super.viewDidLoad()

  locationManager.requestAlwaysAuthorization()

  loadItems()
}
```

- The call to **requestAlwaysAuthorization()** will prompt the user for access to location services if they haven't granted it already.

- **Always** and **When in Use** are variants on location permissions.

- When the user grants **Always** authorization to the app, the app can start any of the available location services while it is running in the foreground or background.

- The point of this app is to monitor for iBeacon regions at all times, so you'll need the **Always** location permissions scope for triggering region events while the app is both in the foreground and background.

- iOS requires that you set up a string value in **Info.plist** that will be displayed to the user when access to their location is required by the app.

- If you don't set this up, location services won't work at all — you don't even get a warning!

- Open **Info.plist** and add a new entry by clicking on the + that appears when you select the **Information Property List** row.

Key		Type		Value
▼ Information Property List	⊕	Dictionary	↕	(14 items)
Localization native development region	↕	String		en
Executable file	▲	String		$(EXECUTABLE_NAME)

- Fortunately, the key you need to add is in the pre-defined list shown in the dropdown list of keys — just scroll down to the **Privacy** section.

- Select the key **Privacy – Location Always Usage Description** and make sure the **Type** is set to **String**.

- Then add the phrase you want to show to the user to tell them why you need location services on, for example: "ForgetMeNot needs to know where you are".

Key		Type	Value
▼ Information Property List		Dictionary	(15 items)
Privacy - Location Always Usage Descript... ⇕ ⊕ ⊖		String	⇕ ForgetMeNot needs to know where you are
Localization native development region	⇕	String	en
Executable file	⌃	String	$(EXECUTABLE_NAME)

- **Build and run** your app; once running, you should be shown a message asking you to allow the app access to your location:

- Select 'Allow', and the app will be able to track your iBeacons.
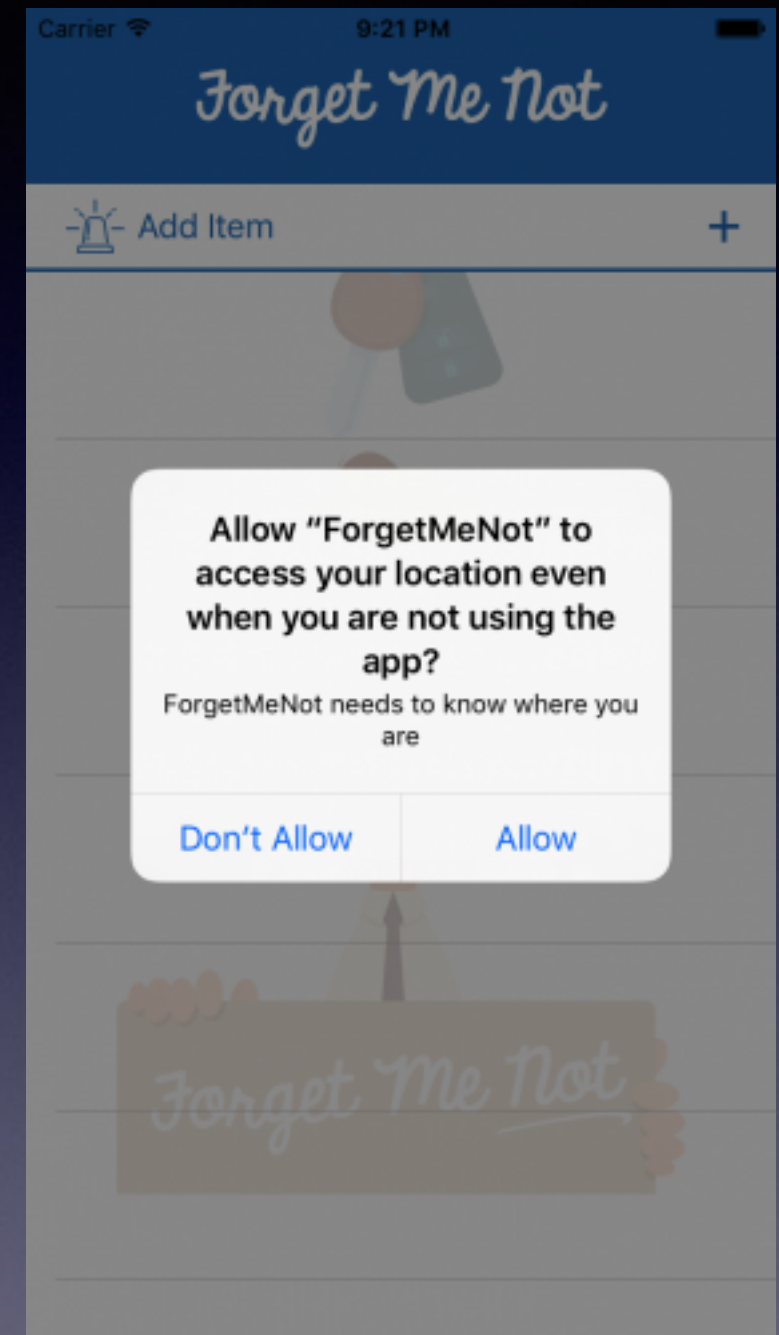
Listening for Your iBeacon

- Now that your app has the location permissions it needs, it's time to find those beacons! Add the following class extension to the bottom of **ItemsViewController.swift** :

```
// MARK: - CLLocationManagerDelegate

extension ItemsViewController: CLLocationManagerDelegate {
}
```

- This will declare **ItemsViewController** as conforming to **CLLocationManagerDelegate**.

- You'll add the delegate methods inside this extension to keep them nicely grouped together.

- Next, add the following line inside of viewDidLoad():

```
locationManager.delegate = self
```

- This sets the **CLLocationManager** delegate to self so you'll receive delegate callbacks.

- Now that your location manager is set up, you can instruct your app to begin monitoring for specific regions using **CLBeaconRegion**.

- When you register a region to be monitored, those regions persist between launches of your application.

- This will be important later when you respond to the boundary of a region being crossed while your application is not running.

- Your iBeacon items in the list are represented by the the **Item** model via the **items** array property.

- **CLLocationManager**, however, expects you to provide a **CLBeaconRegion** instance in order to begin monitoring a region.

- In **Item.swift** create the following helper method on **Item**:

```swift
func asBeaconRegion() -> CLBeaconRegion {
  return CLBeaconRegion(proximityUUID: uuid,
                                major: majorValue,
                                minor: minorValue,
                           identifier: name)
}
```

- This returns a new **CLBeaconRegion** instance derived from the current **Item**.

- You can see that the classes are similar in structure to each other, so creating an instance of **CLBeaconRegion** is very straightforward since it has direct analogs to the UUID, major value, and minor value.

- Now you need a method to begin monitoring a given item.

- Open **ItemsViewController.swift** and add the following method to **ItemsViewController**:

```swift
func startMonitoringItem(_ item: Item) {
  let beaconRegion = item.asBeaconRegion()
  locationManager.startMonitoring(for: beaconRegion)
  locationManager.startRangingBeacons(in: beaconRegion)
}
```

- This method takes an **Item** instance and creates a **CLBeaconRegion** using the method you defined earlier.

- It then tells the location manager to start monitoring the given region, and to start ranging iBeacons within that region.

- Ranging is the process of discovering iBeacons within the given region and determining their distance.

- An iOS device receiving an iBeacon transmission can approximate the distance from the iBeacon.

- The distance (between transmitting iBeacon and receiving device) is categorized into 3 distinct ranges:

  - **Immediate** Within a few centimeters

  - **Near** Within a couple of meters

  - **Far** Greater than 10 meters away

- By default, monitoring notifies you when the region is entered or exited regardless of whether your app is running.

- Ranging, on the other hand, monitors the proximity of the region only while your app is running.

- You'll also need a way to stop monitoring an item's region after it's deleted.

- Add the following method to **ItemsViewController**:

```swift
func stopMonitoringItem(_ item: Item) {
  let beaconRegion = item.asBeaconRegion()
  locationManager.stopMonitoring(for: beaconRegion)
  locationManager.stopRangingBeacons(in: beaconRegion)
}
```

- The above method reverses the effects of **startMonitoringItem(_:)** and instructs the **CLLocationManager** to stop monitor and ranging activities.

- Now that you have the start and stop methods, it's time to put them to use!

- The natural place to start monitoring is when a user adds a new item to the list.

- Have a look at **addBeacon(_:)** in **ItemsViewController.swift**.

- This protocol method is called when the user hits the Add button in **AddItemViewController** and creates a new Item to monitor.

- Find the call to **persistItems()** in that method and add the following line just before it:

```
startMonitoringItem(item)
```

- That will activate monitoring when the user saves an item.

- Likewise, when the app launches, the app loads persisted items from **UserDefaults**, which means you have to start monitoring for them on startup too.

- In **ItemsViewController.swift**, find **loadItems()** and add the following line inside the **for** loop at the end:

```
startMonitoringItem(item)
```

- This will ensure each item is being monitored.

- Now you need to take care of removing items from the list.

- Find **tableView(_:commit:forRowAt:)** and add the following line inside the **if** statement:

```
stopMonitoringItem(items[indexPath.row])
```

- This table view delegate method is called when the user deletes the row.

- The existing code handles removing it from the model and the view, and the line of code you just added will also stop the monitoring of the item.

- At this point you've made a lot of progress!

- Your application now starts and stops listening for specific iBeacons as appropriate.

- You can build and run your app at this point; but even though your registered iBeacons might be within range your app has no idea how to react when it finds one…time to fix that!

Acting on Found iBeacons

- Now that your location manager is listening for iBeacons, it's time to react to them by implementing some of the **CLLocationManagerDelegate** methods.

- First and foremost is to add some error handling, since you're dealing with very specific hardware features of the device and you want to know if the monitoring or ranging fails for any reason.

- Add the following two methods to the **CLLocationManagerDelegate** class extension you defined earlier at the bottom of **ItemsViewController.swift**:

```swift
func locationManager(_ manager: CLLocationManager, monitoringDidFailFor region: CLRegion?, withError error: Error) {
  print("Failed monitoring region: \(error.localizedDescription)")
}

func locationManager(_ manager: CLLocationManager, didFailWithError error: Error) {
  print("Location manager failed: \(error.localizedDescription)")
}
```

- These methods will simply log any received errors as a result of monitoring iBeacons.

- If everything goes smoothly in your app you should never see any output from these methods.

- However, it's possible that the log messages could provide very valuable information if something isn't working.

- The next step is to display the perceived proximity of your registered iBeacons in real-time.

- Add the following stubbed-out method to the **CLLocationManagerDelegate** class extension:

```swift
func locationManager(_ manager: CLLocationManager, didRangeBeacons beacons: [CLBeacon], in region:
CLBeaconRegion) {

  // Find the same beacons in the table.
  var indexPaths = [IndexPath]()
  for beacon in beacons {
    for row in 0..<items.count {
        // TODO: Determine if item is equal to ranged beacon
    }
  }

  // Update beacon locations of visible rows.
  if let visibleRows = tableView.indexPathsForVisibleRows {
    let rowsToUpdate = visibleRows.filter { indexPaths.contains($0) }
    for row in rowsToUpdate {
      let cell = tableView.cellForRow(at: row) as! ItemCell
      cell.refreshLocation()
    }
  }
}
```

- This delegate method is called when iBeacons come within range, move out of range, or when the range of an iBeacon changes.

- The goal of your app is to use the array of ranged iBeacons supplied by the delegate methods to update the list of items and display their perceived proximity.

- You'll start by iterating over the **beacons** array, and then iterating over **items** to see if there are matches between in-range iBeacons and the ones in your list.

- Then the bottom portion updates the location string for visible cells.

- You'll come back to the TODO section in just a moment.

- Open **Item.swift** and add the following property to the **Item** class:

```
var beacon: CLBeacon?
```

- This property stores the last **CLBeacon** instance seen for this specific item, which is used to display the proximity information.

- Now add the following equality operator at the bottom of the file, outside the class definition:

```swift
func ==(item: Item, beacon: CLBeacon) -> Bool {
  return ((beacon.proximityUUID.uuidString == item.uuid.uuidString)
        && (Int(beacon.major) == Int(item.majorValue))
        && (Int(beacon.minor) == Int(item.minorValue)))
}
```

- This equality function compares a **CLBeacon** instance with an **Item** instance to see if they are equal — that is, if all of their identifiers match.

- In this case, a **CLBeacon** is equal to an **Item** if the UUID, major, and minor values are all equal.

- Now you'll need to complete the ranging delegate method with a call to the above helper method.

- Open **ItemsViewController.swift** and return to **locationManager(_:didRangeBeacons:inRegion:)**.

- Replace the **TODO** comment in the innermost **for** loop with the following:

```
if items[row] == beacon {
  items[row].beacon = beacon
  indexPaths += [IndexPath(row: row, section: 0)]
}
```

- Here, you set the cell's **beacon** when you find a matching item and iBeacon. Checking that the item and beacon match is easy thanks to your equality operator!

- Each CLBeacon instance has a proximity property which is an enum with values of far, near, immediate, and unknown.

- Add the following method to **Item**:

```swift
func nameForProximity(_ proximity: CLProximity) -> String {
  switch proximity {
  case .unknown:
    return "Unknown"
  case .immediate:
    return "Immediate"
  case .near:
    return "Near"
  case .far:
    return "Far"
  }
}
```

- This returns a human-readable proximity value from **proximity** which you'll use next.

- Still in **Item**, add the following method:

```swift
func locationString() -> String {
  guard let beacon = beacon else { return "Location: Unknown" }
  let proximity = nameForProximity(beacon.proximity)
  let accuracy = String(format: "%.2f", beacon.accuracy)

  var location = "Location: \(proximity)"
  if beacon.proximity != .unknown {
    location += " (approx. \(accuracy)m)"
  }

  return location
}
```

- This generates a nice, neat string describing not only the proximity range of the beacon, but also the approximate distance.

- Now it's time to use that new method to display the perceived proximity of the ranged iBeacon.

- Open **ItemCell.swift** and add the following to just below the **lblName.text = item.name** line of code:

```
lblLocation.text = item.locationString()
```

- This displays the location for each cell's beacon.

- And to ensure it shows updated info, add the following inside **refreshLocation()**:

```
lblLocation.text = item?.locationString() ?? ""
```

- **refreshLocation()** is called each time the **locationManager** ranges the beacon, which sets the cell's **lblLocation.text** property with the perceived proximity value and approximate 'accuracy' taken from the **CLBeacon**.

- This latter value may fluctuate due to RF interference even when your device and iBeacon are not moving, so don't rely on it for a precise location for the beacon.

- Now ensure your iBeacon is registered and move your device closer or away from your device.

- You'll see the label update as you move around, as shown below:
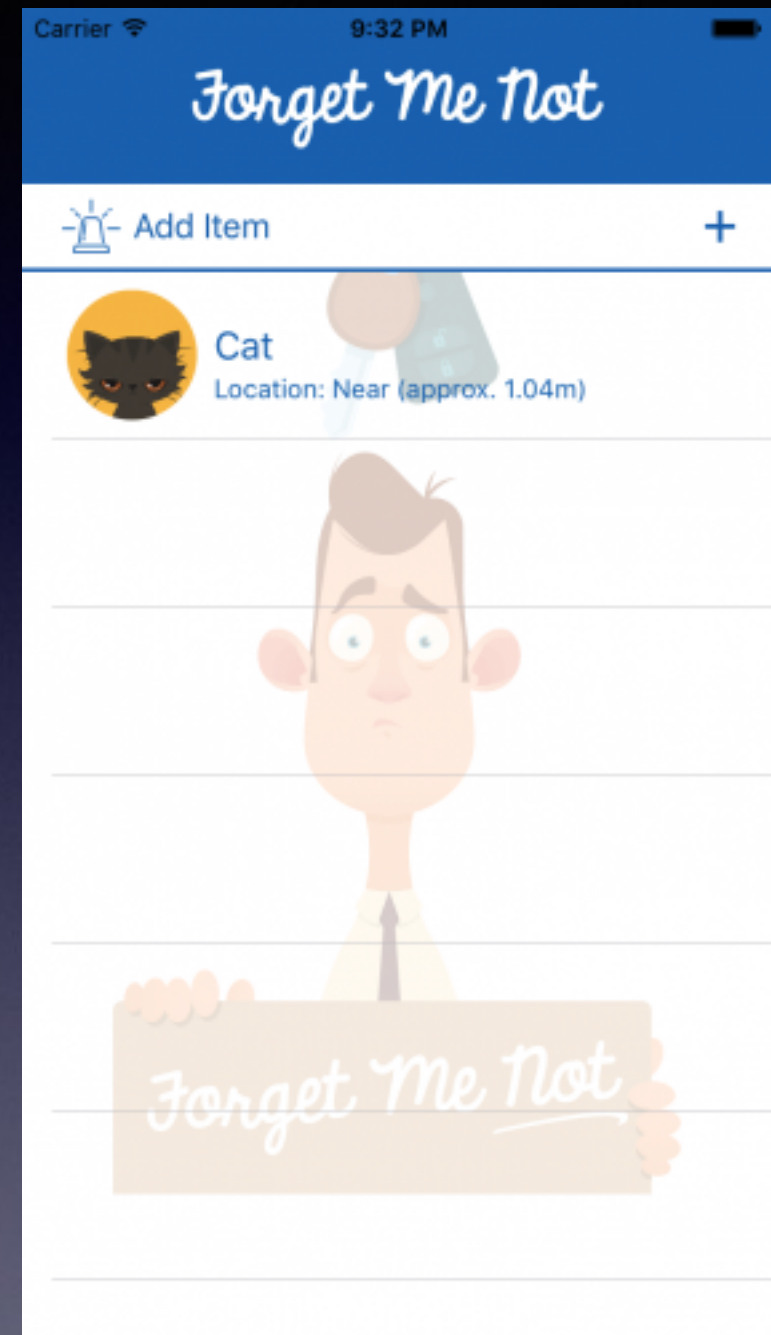
- You may find that the perceived proximity and accuracy is drastically affected by the physical location of your iBeacon; if it is placed inside of something like a box or a bag, the signal may be blocked as the iBeacon is a very low-power device and the signal may easily become attenuated.

- Keep this in mind when designing your application — and when deciding the best placement for your iBeacon hardware.

Notifications

- Things feel pretty complete at this point; you have your list of iBeacons and can monitor their proximity in real time.

- But that isn't the end goal of your app.

- You still need to notify the user when the app is not running in case they forgot their laptop bag or their cat ran away — or worse, if their cat ran away with the laptop bag! :]

- They look so innocent, don't they?

- At this point, you've probably noticed it doesn't take much code to add iBeacon functionality to your app.

- Adding a notification when a cat runs away with your laptop bag is no different!

- Open **AppDelegate.swift** and add the following import:

```swift
import CoreLocation
```

- Next, make the **AppDelegate** class conform to the **CLLocationManagerDelegate** protocol by adding the following to the very bottom of **AppDelegate.swift** (below the closing brace):

```
// MARK: - CLLocationManagerDelegate
extension AppDelegate: CLLocationManagerDelegate {

}
```

- Just as before, you need to initialize the location manager and set the delegate accordingly.

- Add a new **locationManager** property to the **AppDelegate** class, initialized with an instance of **CLLocationManager**:

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

- Then add the following statement to the very top of **application(_:didFinishLaunchingWithOptions:)**:

```
locationManager.delegate = self
```

- Recall that any regions you add for monitoring using **startMonitoringForRegion(_:)** are shared by all location managers in your application.

- So the final step here is simply to react when Core Location wakes up your app when a region is encountered.

- Add the following method to the class extension you added at the bottom of **AppDelegate.swift**, like so:

```swift
func locationManager(_ manager: CLLocationManager, didExitRegion region: CLRegion) {
  guard region is CLBeaconRegion else { return }

  let content = UNMutableNotificationContent()
  content.title = "Forget Me Not"
  content.body = "Are you forgetting something?"
  content.sound = .default()

  let request = UNNotificationRequest(identifier: "ForgetMeNot", content: content, trigger: nil)
  UNUserNotificationCenter.current().add(request, withCompletionHandler: nil)
}
```

- Your location manager calls the above method when you exit a region, which is the event of interest for this app.

- You don't need to be notified if you move closer to your laptop bag — only if you move too far away from it.

- Here you check the region to see if it's a **CLBeaconRegion**, since it's possible it could be a **CLCircularRegion** if you're also performing geolocation region monitoring.

- Then you post a local notification with the generic message "Are you forgetting something?".

- In iOS 8 and later, apps that use either local or remote notifications must register the types of notifications they intend to deliver.

- The system then gives the user the ability to limit the types of notifications your app displays.

- The system does not badge icons, display alert messages, or play alert sounds if any of these notification types are not enabled for your app, even if they are specified in the notification payload.

- Add the following to the top of of **application(_:didFinishLaunchingWithOptions:)**:

```swift
// Request permission to send notifications
let center = UNUserNotificationCenter.current()
center.requestAuthorization(options:[.alert, .sound]) { (granted, error) in }
```

- This simply says that the app wishes to display an alert and play a sound when it receives a notification.

- Build and run your app; make sure that your app can see one of your registered iBeacons and put the app into the background by pressing the Home button — which is a real-world scenario given that you want the app to notify you whilst you're pre-occupied with something else — perhaps another Ray Wenderlich tutorial app? :].

- Now move away from the iBeacon and once you're far enough away you'll see the notification pop up:
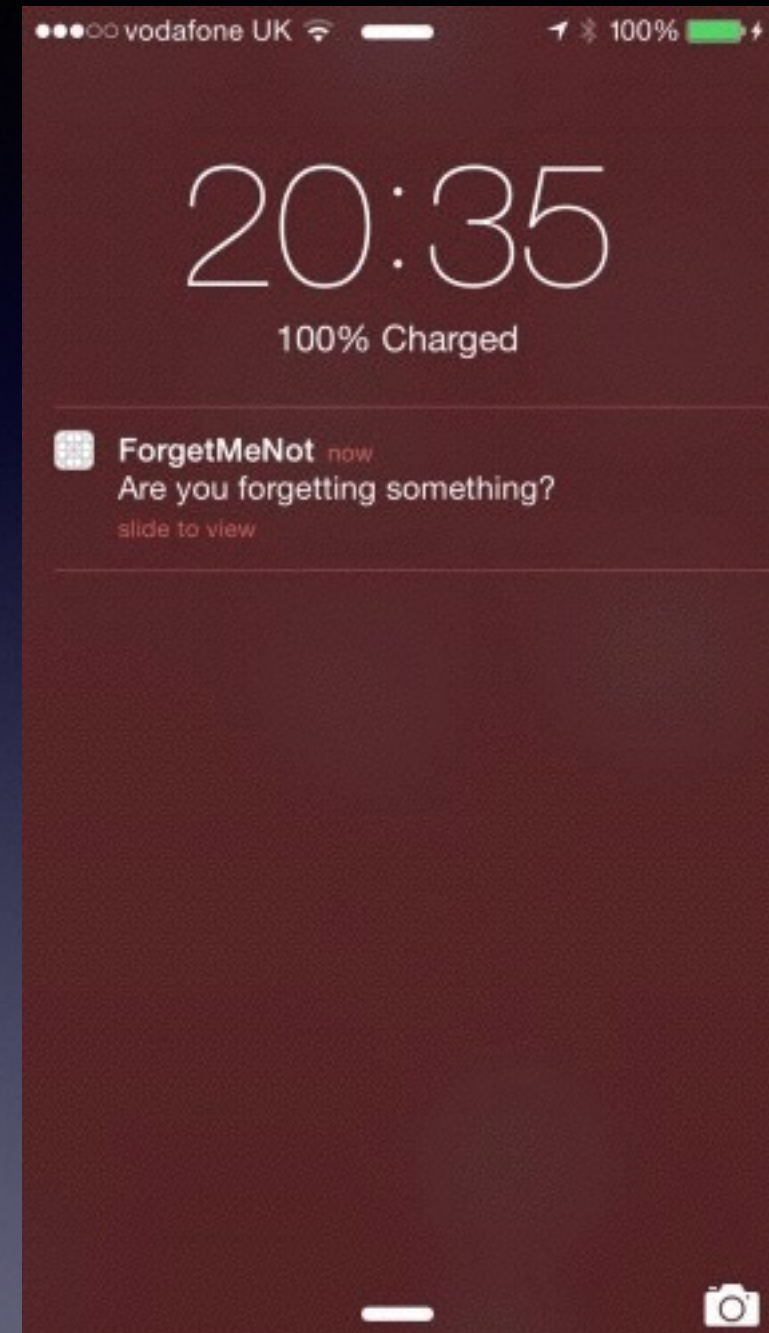
Reference

- https://www.raywenderlich.com/152330/ibeacon-tutorial-ios-swift?utm_source=raywenderlich.com+Weekly&utm_campaign=0ae7d23ab3-raywenderlich_com_Weekly_Issue_115&utm_medium=email&utm_term=0_83b6edc87f-0ae7d23ab3-415659401

- 講義、範例程式下載：

- https://github.com/ ycwang812/UCH