MapKit Tutorial: Getting Started

Learn how to use the MapKit framework to display real-world points of interest in your own apps.

By Audrey Tam Jun 27 2017 · Article (35 mins) · Beginner

4.3/5

6 Ratings · Leave a Rating

Update June 27, 2017: Updated by Audrey Tam for Xcode 9 beta / iOS 11 / Swift 4. Original post by Ray Wenderlich.

MapKit is a really useful API available on iOS devices that makes it easy to display maps, plot locations, and even draw routes and other shapes on top.

This update uses public artworks data from Honolulu, where I was born and raised. It's no longer my hometown, but the names and places bring back memories. If you're not lucky enough to *live* there, I hope you'll enjoy imagining yourself being there!

In this tutorial, you'll make an app that zooms into a location in Honolulu, and plot one of the artworks on the map. You'll implement the pin's callout detail button to launch the *Maps* app, with driving/walking directions to the artwork. Your app will then parse a JSON file from a Honolulu data portal, to extract the public artwork objects, and plot them on the map.

In the process, you'll learn how to add a *MapKit* map to your app, zoom to a particular location, parse government data that uses the Socrata Framework, create custom map annotations, and more!

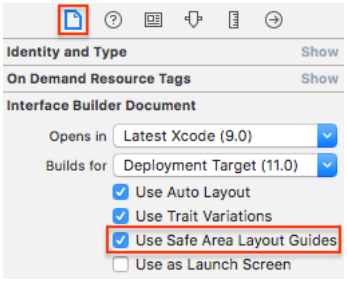
This tutorial assumes some familiarity with Swift and iOS programming. If you are a complete beginner, check out some of the other tutorials on this site.

Now let's get mapping!

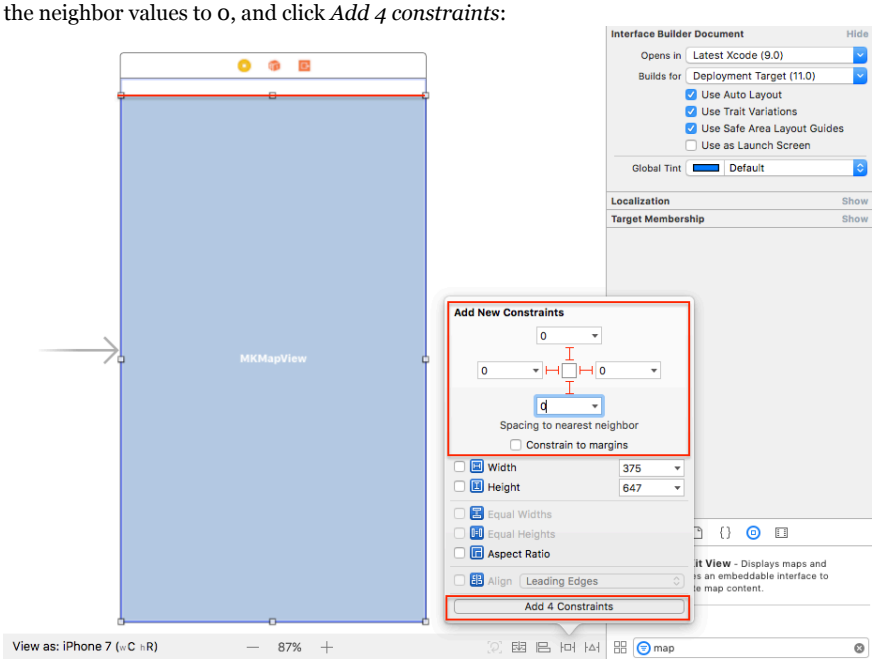
Getting Started

Start by downloading the starter project, which contains the JSON file and some image assets, but no maps yet!

Open *Main.storyboard*. In the *File Inspector*, check the box for Use Safe Area Layout Guides. This stops you setting constraints relative to the deprecated layout guides, and that stops "deprecated" warnings.



In the *Document Outline*, select *Safe Area*, to see its top edge is slightly lower than the view's top edge. From the *Object library*, drag a *MapKit View* into the upper corner of the scene, aligning its top edge with the dashed blue line *below* the view's top edge, then drag its lower right corner to meet the view's lower right corner. Use the *Add New Constraints* auto layout menu (the TIE fighter icon) to pin the map view: uncheck *Constrain to margins*, then set all the neighbor values to 0, and click *Add 4 constraints*:



Note: Normally, you don't have to manually stretch the map view onto the scene — simply use the *Add New Constraints* menu to pin its edges — but this isn't yet working in Xcode 9 beta.

Next, add this line to *ViewController.swift*, just below the `import UIKit` statement:

```
import MapKit
```

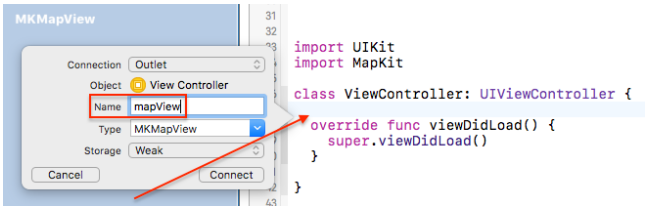
Build and run your project, and you'll have a fully zoomable and pannable map showing the continent of your current location, using Apple Maps!

So far so good, eh? But you don't want to start the map looking at the entire world – you want to zoom into a particular area!

To control the map view, you must create an *outlet* for it in *ViewController.swift*.

In the storyboard, open the *assistant editor*: it should display *ViewController.swift*.

To create the outlet, click the *Map View* in *Main.storyboard*, and control-drag from it into the space just inside the `ViewController` class definition: Xcode should prompt you to *Insert Outlet or Outlet Collection*. Release the drag and, in the pop-up window, name the outlet `mapView`:



Xcode adds a `mapView` property to the `ViewController` class: you'll use this to control what the map view displays.

Setting Visible Area

Switch back to the *standard editor* and, in *ViewController.swift*, find `viewDidLoad()`, and add the following to the end of the method:

```
// set initial location in Honolulu
let initialLocation = CLLocation(latitude: 21.282778, longitude: -157.829444)
```

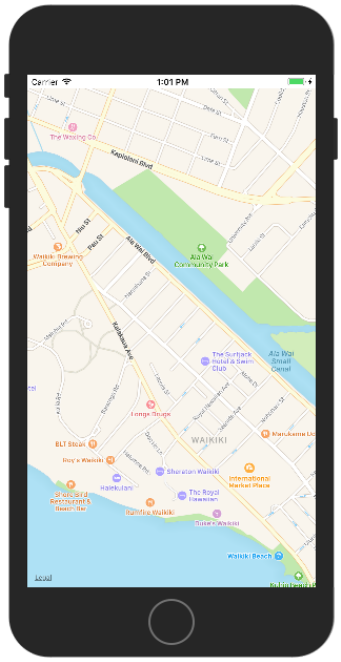
You'll use this to set the starting coordinates of the map view to a point in Honolulu.
When telling the map what to display, giving a latitude and longitude is enough to center the map, but you must also specify the rectangular *region* to display, to get a correct zoom level.
Add the following constant and helper method to the class:

```
let regionRadius: CLLocationDistance = 1000
func centerMapOnLocation(location: CLLocation) {
  let coordinateRegion = MKCoordinateRegionMakeWithDistance(location.coordinate,
    regionRadius, regionRadius)
  mapView.setRegion(coordinateRegion, animated: true)
}
```

The `location` argument is the center point. The region will have north-south and east-west spans based on a distance of `regionRadius`. You set this to `1000` meters: a little more than half a mile, which works well for plotting the public artwork data in the JSON file.
`setRegion(_:animated:)` tells `mapView` to display the region. The map view automatically transitions the current view to the desired region with a neat zoom animation, with no extra code required!
Back in `viewDidLoad()`, add the following line to the end of the method:

```
centerMapOnLocation(location: initialLocation)
```

You're calling the helper method to zoom into `initialLocation` on startup.
Build and run the app, and you'll find yourself in the heart of Waikiki: aloha! :]



Obtaining Public Artworks Data

The next step is to plot interesting data around the current location. But where in the world can we get such stuff?
Well, it depends on your current location. Honolulu, like many cities, has an Open Data Portal to improve public access to government data. Like many cities, Honolulu's data portal is "Powered by Socrata", an open data framework that provides a rich set of developer tools for accessing Socrata-based data.
After you finish this tutorial, maybe look around to see if a nearby city has an alternate dataset you can use?
For this tutorial, you'll be using the Honolulu Public Art dataset. To keep things simple, I've already downloaded this data from the portal, and included it in the starter project.
To get a feeling for the items in this dataset, open *PublicArt.json* in the Xcode editor and scroll down to line 1180 (or use ⌘ + L for *Jump to Line*), which begins with `"data"` followed by an array of arrays – one array for each artwork. For this tutorial, you'll use only a few properties from each array: the artwork's location name, discipline, title, latitude and longitude. For example, for the first data item:
location name: Lester McCoy Pavilion

discipline: Mural
title: The Makahiki Festival – The Makai Mural
latitude: 21.290824
longitude: -157.85131
Later in this tutorial, you'll parse this dataset to create an array of artworks but first, to jump straight into the *MapKit* fun, you'll just plot *one* of the artworks on the map.

Showing an Artwork on the Map

In *PublicArt.json*, jump or scroll to item 55 at line 1233: it's a bronze statue of King David Kalakaua in Waikiki Gateway Park – ah, can you hear the waves breaking on the beach?

Photo of King David Kalakaua statue, by Wally Gobetz

The properties for this item are:
location name: Waikiki Gateway Park
discipline: Sculpture
title: King David Kalakaua
latitude: 21.283921
longitude: -157.831661
To show this on the map view, you must create a *map annotation*. Map annotations are small pieces of information tied to a particular location, and are often represented in Apple's Maps app as little pins.
To create your own annotations, you create a class that conforms to the `MKAnnotation` protocol, add the annotation to the map, and inform the map how the annotation should be displayed.

The Artwork Class

First, create an `Artwork` class in a new Swift file: *File\New\File*, choose *iOS\Source\Swift File*, and click *Next*. Set the *Save As* field to *Artwork.swift* and click *Create*.
Open *Artwork.swift* in the editor and add the following, below `import Foundation`:

```
import MapKit

class Artwork: NSObject, MKAnnotation {
  let title: String?
  let locationName: String
  let discipline: String
  let coordinate: CLLocationCoordinate2D

  init(title: String, locationName: String, discipline: String, coordinate: CLLocationCoordinate2D) {
    self.title = title
    self.locationName = locationName
    self.discipline = discipline
    self.coordinate = coordinate

    super.init()
  }

  var subtitle: String? {
    return locationName
  }
}
```

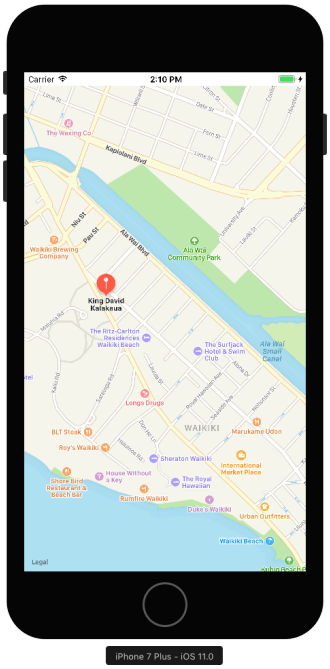
To adopt the `MKAnnotation` protocol, `Artwork` must subclass `NSObject`, because `MKAnnotation` is an `NSObjectProtocol`.
The `MKAnnotation` protocol requires the `coordinate` property. If you want your annotation view to display a title and subtitle when the user taps a pin, your class also needs properties named `title` and `subtitle`.
It's perfectly sensible for the `Artwork` class to have stored properties named `title` and `coordinate`, but none of the *PublicArt.json* properties maps naturally to the idea of "subtitle". To conform to the `MKAnnotation` protocol, you make `subtitle` a computed property that returns `locationName`.
OK, so the `title`, `locationName` and `coordinate` properties will be used for the `MKAnnotation` object, but what's the `discipline` property for? You'll find out later in this tutorial! ;]

Adding an Annotation

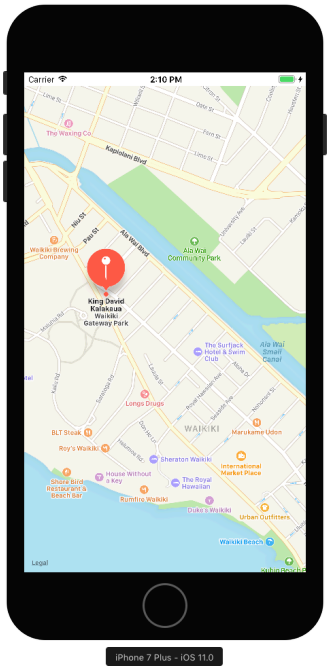
Next, you'll add an `Artwork` object to the map view, for every artwork you want to plot. For now, you're adding only one artwork, so switch to *ViewController.swift* and add the following lines to the end of `viewDidLoad()`:

```
// show artwork on map
let artwork = Artwork(title: "King David Kalakaua",
  locationName: "Waikiki Gateway Park",
  discipline: "Sculpture",
  coordinate: CLLocationCoordinate2D(latitude: 21.283921, longitude: -157.831661))
mapView.addAnnotation(artwork)
```

Here, you create a new `Artwork` object, and add it as an annotation to the map view. The `MKMapView` class also has an `addAnnotations:` (plural) method, which you'll use later in this tutorial, when you have an array of annotations to add to the map view.
Build and run your project, and now you should see where King David Kalakaua's statue is, at the gateway to Waikiki!



The default annotation marker view shows the location, with the title below the marker. Select the marker: it grows, and now shows the subtitle, as well:



Well, that's ok, but you're used to pins that show a callout — a little bubble — when the user taps the marker. For that, you must configure the annotation view, and that's the next step.

Configuring the Annotation View

One way to configure the annotation view is to implement the map view's `mapView(_:viewFor:)` delegate method. Your job in this delegate method is to return an instance of `MKAnnotationView`, to present as a visual indicator of the annotation.
In this case, `ViewController` will be the delegate for the map view. To avoid clutter and improve readability, you'll create an *extension* of the `ViewController` class.
Add the following at the bottom of *ViewController.swift*:

```
extension ViewController: MKMapViewDelegate {
  // 1
  func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {
    // 2
    guard let annotation = annotation as? Artwork else { return nil }
    // 3
    let identifier = "marker"
    var view: MKMarkerAnnotationView
    // 4
    if let dequeuedView = mapView.dequeueReusableAnnotationView(withIdentifier: identifier)
      as? MKMarkerAnnotationView {
      dequeuedView.annotation = annotation
      view = dequeuedView
    } else {
      // 5
      view = MKMarkerAnnotationView(annotation: annotation, reuseIdentifier: identifier)
      view.canShowCallout = true
      view.calloutOffset = CGPoint(x: -5, y: 5)
      view.rightCalloutAccessoryView = UIButton(type: .detailDisclosure)
    }
    return view
  }
}
```

Here's what you're doing:
`mapView(_:viewFor:)` gets called for every annotation you add to the map (just like `tableView(_:cellForRowAt:)` when working with table views), to return the view for each annotation.

Your app might use other annotations, like user location, so check that this annotation is an `Artwork` object. If it isn't, return `nil` to let the map view use its default annotation view.

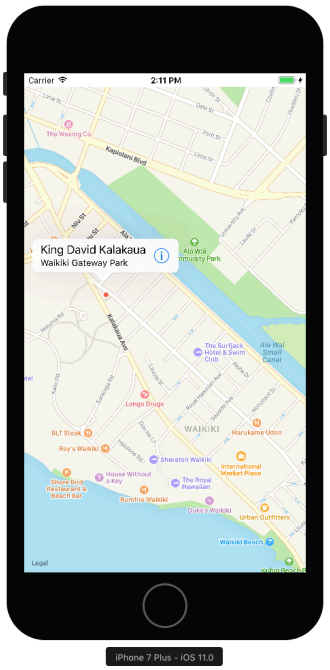
To make markers appear, you create each view as an `MKMarkerAnnotationView`. Later in this tutorial, you'll create `MKAnnotationView` objects, to display images instead of markers.

Also similarly to `tableView(_:cellForRowAt:)`, a map view reuses annotation views that are no longer visible. So you check to see if a reusable annotation view is available before creating a new one.

Here you create a new `MKMarkerAnnotationView` object, if an annotation view could not be dequeued. It uses the title and subtitle properties of your `Artwork` class to determine what to show in the callout.
Note: One extra thing to point out about this, suggested by Kalgar, when you dequeue a reusable annotation, you give it an identifier. If you have multiple styles of annotations, be sure to have a unique identifier for each one, otherwise you might mistakenly dequeue an identifier of a different type, and have unexpected behavior in your app. Again, it's the same idea behind a cell identifier in `tableView(_:cellForRowAt:)`.
All that's left is setting `ViewController` as the delegate of the map view. You can do this in *Main.storyboard*, but I prefer to do it in code, where it's more visible. In *ViewController.swift*, add this line to `viewDidLoad()`, before the statement that creates `artwork`:

```
mapView.delegate = self
```

And that's it! Build and run your project, and tap the marker to pop up the callout bubble:



`mapView(_:viewFor:)` configures the callout to include a *detail disclosure* info button on the right side but tapping that button doesn't do anything yet. You could implement it to show an alert with more info, or to open a detail view controller.
Here's a neat third option: when the user taps the info button, your app will launch the *Maps* app, complete with driving/walking/transit directions to get from the simulated user location to the artwork!

Launching the Maps App

To provide this great user experience, open *Artwork.swift* and add this `import` statement, below the other two:

```
import Contacts
```

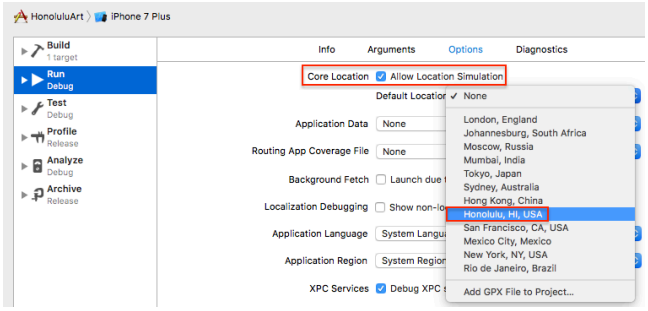
This adds the *Contacts* framework, which contains dictionary key constants such as `CNPostalAddressStreetKey`, for when you need to set the address, city or state fields of a location.
Next, add the following helper method to the class:

```
// Annotation right callout accessory opens this mapItem in Maps app
func mapItem() -> MKMapItem {
  let addressDict = [CNPostalAddressStreetKey: subtitle!]
  let placemark = MKPlacemark(coordinate: coordinate, addressDictionary: addressDict)
  let mapItem = MKMapItem(placemark: placemark)
  mapItem.name = title
  return mapItem
}
```

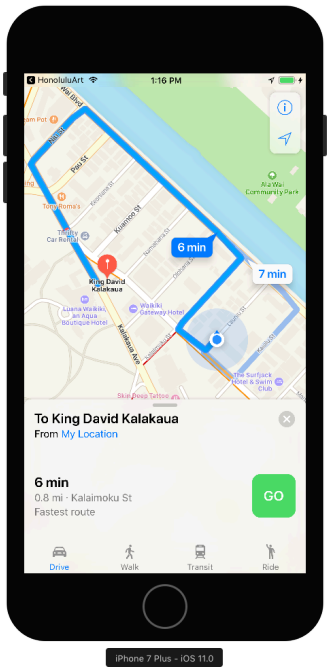
Here you create an `MKMapItem` from an `MKPlacemark`. The Maps app is able to read this `MKMapItem`, and display the right thing.
Next, you have to tell MapKit what to do when the user taps the callout button. Open *ViewController.swift*, and add this method to the `MKMapViewDelegate` extension:

```
func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView,
    calloutAccessoryControlTapped control: UIControl) {
  let location = view.annotation as! Artwork
  let launchOptions = [MKLaunchOptionsDirectionsModeKey: MKLaunchOptionsDirectionsModeDriving]
  location.mapItem().openInMaps(launchOptions: launchOptions)
}
```

When the user taps a map annotation marker, the callout shows an info button. If the user taps this info button, the `mapView(_:annotationView:calloutAccessoryControlTapped:)` method is called.
In this method, you grab the `Artwork` object that this tap refers to, and then launch the Maps app by creating an associated `MKMapItem`, and calling `openInMaps(launchOptions:)` on the map item.
Notice you're passing a dictionary to this method. This allows you to specify a few different options; here the `DirectionModeKey` is set to `Driving`. This causes the Maps app to show driving directions from the user's current location to this pin. Neat!
Note: Explore the `MKMapItem` documentation to see other launch option dictionary keys, and the `openMaps(with:launchOptions:)` method that lets you pass an array of `MKMapItem` objects.
Before you build and run, you should move to Honolulu – well, actually, just set your simulated location to Honolulu. In Xcode, go to *Product\Scheme\Edit Scheme...*, select *Run* from the left menu, then select the *Options* tab. Check *Core Location: Allow Location Simulation*, and select *Honolulu, HI, USA* as the *Default Location*. Then click the *Close* button:



Build and run the app, and you'll see the map zoom in on Waikiki, as before. Tap on the marker, then tap the info button in the callout, and watch it launch the *Maps* app to show the statue's location, with driving directions to it:



This calls for a celebration with your favorite tropical drink!
Note: The first time you open Maps, it prompts you to allow Maps to access your location (tap Allow), and displays a Safety Warning.

Parsing JSON Data into Artwork Objects

Now that you know how to show one artwork on the map, and how to launch the *Maps* app from the pin's callout info button, it's time to parse the dataset into an array of `Artwork` objects. Then you'll add them as annotations to the map view, to display all artworks located in the current map region.
Add this failable initializer to *Artwork.swift*, below the initializer:

```
init?(json: [Any]) {
  // 1
  self.title = json[16] as? String ?? "No Title"
  self.locationName = json[12] as! String
  self.discipline = json[15] as! String
  // 2
  if let latitude = Double(json[18] as! String),
    let longitude = Double(json[19] as! String) {
  self.coordinate = CLLocationCoordinate2D(latitude: latitude, longitude: longitude)
  } else {
    self.coordinate = CLLocationCoordinate2D()
  }
}
```

Here's what you're doing:
The `json` argument is one of the arrays that represent an artwork – an array of `Any` objects. If you count through an array's elements, you'll see that the `title`, `locationName` etc. are at the indexes specified in this method. The `title` field for some of the artworks is `null`, so you provide a default value for the `title` value.

The latitude and longitude values in the `json` array are strings: if you can create `Double` objects from them, you create a `CLLocationCoordinate2D`.
In other words, this initializer converts an array like this:

```
[ 55, "8492E480-43E9-4683-927F-0E82F3E1A024", 55, 1340413921, "436621", 1340413921, "436621", "{\n}", "Sean Browne", "Gift
```

into an `Artwork` object like the one you created before:
`locationName`: "Waikiki Gateway Park"
`discipline`: "Sculpture"
`title`: "King David Kalakaua"

`coordinate` with latitude: 21.283921 longitude: -157.831661
To use this initializer, open *ViewController.swift*, and add the following property to the class – an array to hold the `Artwork` objects from the JSON file:

```
var artworks: [Artwork] = []
```

Next, add the following helper method to the class:

```
func loadInitialData() {
  // 1
  guard let fileName = Bundle.main.path(forResource: "PublicArt", ofType: "json")
    else { return }
  let optionalData = try? Data(contentsOf: URL(fileURLWithPath: fileName))

  guard
    let data = optionalData,
    // 2
    let json = try? JSONSerialization.jsonObject(with: data),
    // 3
    let dictionary = json as? [String: Any],
    // 4
    let works = dictionary["data"] as? [[Any]]
    else { return }
  // 5
  let validWorks = works.flatMap { Artwork(json: $0) }
  artworks.append(contentsOf: validWorks)
}
```

Here's what you're doing in this code:
You read the *PublicArt.json* file into a `Data` object.

You use `JSONSerialization` to obtain a JSON object.

You check that the JSON object is a dictionary with `String` keys and `Any` values.

You're only interested in the JSON object whose key is `"data"`.

You flatmap this array of arrays, using the failable initializer that you just added to the `Artwork` class, and append the resulting `validWorks` to the `artworks` array.

Plotting the Artworks

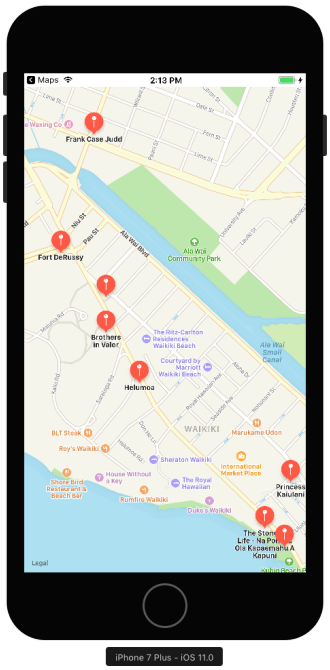
You now have an array of all the public artworks in the dataset, which you'll add to the map.
Still in *ViewController.swift*, add the following code at the end of `viewDidLoad()`:

```
loadInitialData()
mapView.addAnnotations(artworks)
```

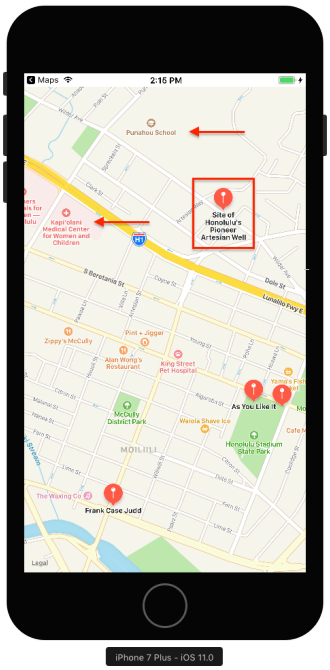
Note: Be sure to use the *plural* `addAnnotations`, not the singular `addAnnotation`!
Comment out or delete the lines that create the single "King David Kalakaua" map annotation – you don't need them, now that `loadInitialData` creates the `artworks` array:

```
//    let artwork = Artwork(title: "King David Kalakaua",
//      locationName: "Waikiki Gateway Park",
//      discipline: "Sculpture",
//      coordinate: CLLocationCoordinate2D(latitude: 21.283921, longitude: -157.831661))
//    mapView.addAnnotation(artwork)
```

Build and run your app and check out all the markers!



Move the map around to see other markers appear. For example, north of your initial location, above Highway 1, is *Honolulu's Pioneer Artesian Well*:



Note: Northwest of the marker is Punahou School, which claims a former US President as an alumnus! And West of the marker is the hospital where he was born. ;]

Tap a marker to open its callout bubble, then tap its info button to launch the *Maps* app – yes, everything you did with the King Kalakaua statue works with all these new artworks!

Note: Thanks to Dave Mark for pointing out that Apple recommends adding all the annotations right away, whether or not they're visible in the map region – when you move the map, it automatically displays the visible annotations.

And that's it! You've built an app that parses a JSON file into an array of artworks, then displays them as annotation markers, with a callout info button that launches the *Maps* app – celebrate with a hula dance around your desk! :]

But wait, there are a few bits of bling that I saved for last...

Customizing Annotations

Markers with Color-Coding & Text

Remember the `discipline` property in the `Artwork` class? Its values are things like "Sculpture" and "Mural" – in fact, the most numerous disciplines are Sculpture, Plaque, Mural and Monument. It's easy to color-code the markers so these disciplines have markers of different colors, with green markers for all the other disciplines.

In *Artwork.swift*, add this property:

```
// markerTintColor for disciplines: Sculpture, Plaque, Mural, Monument, other
var markerTintColor: UIColor  {
  switch discipline {
  case "Monument":
    return .red
```

```
    case "Mural":
      return .cyan
    case "Plaque":
      return .blue
    case "Sculpture":
      return .purple
    default:
      return .green
    }
  }
```

Now, you *could* keep adding code to `mapView(_:viewFor:)`, but that would clutter up the view controller. There's a more elegant way, similar to what you can do for table view cells. Create a new Swift file named *ArtworkViews.swift*, and add this code, below the `import` statement:

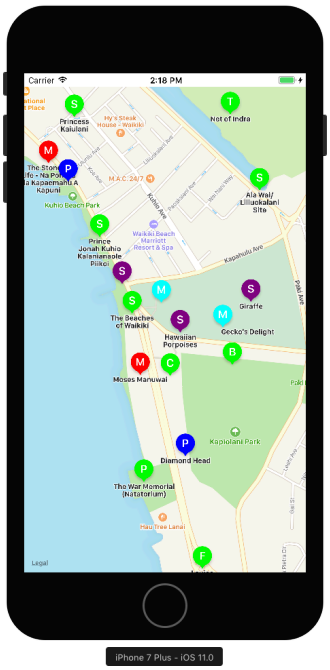
```
import MapKit

class ArtworkMarkerView: MKMarkerAnnotationView {
  override var annotation: MKAnnotation? {
    willSet {
      // 1
      guard let artwork = newValue as? Artwork else { return }
      canShowCallout = true
      calloutOffset = CGPoint(x: -5, y: 5)
      rightCalloutAccessoryView = UIButton(type: .detailDisclosure)
      // 2
      markerTintColor = artwork.markerTintColor
      glyphText = String(artwork.discipline.first!)
    }
  }
}
```

Soon, you'll register this class as a reusable annotation view for `Artwork` annotations. The system will pass it an annotation as `newValue`, so here's what you're doing:
These lines do the same thing as your `mapView(_:viewFor:)`, configuring the callout.

Then you set the marker's tint color, and also replace its pin icon (glyph) with the first letter of the annotation's discipline.
Now switch to *ViewController.swift*, and add this line to `viewDidLoad()`, just before calling `loadInitialData()`:

```
mapView.register(ArtworkMarkerView.self,
  forAnnotationViewWithReuseIdentifier: MKMapViewDefaultAnnotationViewReuseIdentifier)
```

Here, you register your new class with the map view's default reuse identifier. For an app with more annotation types, you would register classes with custom identifiers.
Scroll down to the extension, and comment out the `mapView(_:viewFor:)` method.
Build and run your app, then move the map around, to see the different colored and labeled markers:



In this section of the map, there are actually a lot more artworks than the map view shows: it reduces clutter by clustering markers that are too close together. In the next section, you'll see all the annotations.
But first, set the glyph's image instead of its text. Add the following property to *Artwork.swift*:

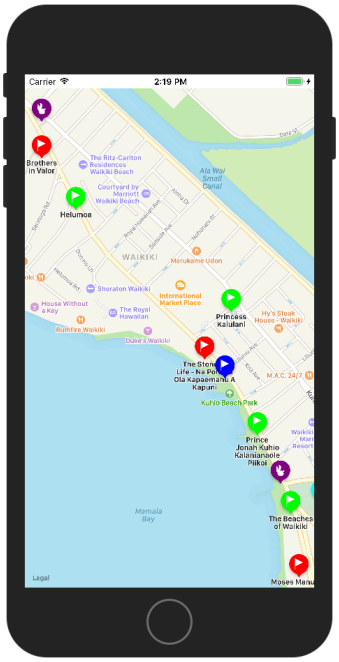
```
var imageName: String? {
  if discipline == "Sculpture" { return "Statue" }
```

```
    return "Flag"
  }
```

These images from icons8.com are already in *Images.xcassets*.
Then, in `ArtworkMarkerView`, comment out the `glyphText` line, and add these lines:

```
if let imageName = artwork.imageName {
  glyphImage = UIImage(named: imageName)
} else {
  glyphImage = nil
}
```

These images from icons8.com are already in *Images.xcassets*.
Build and run your app to see different colored markers with images:



And that's a segue to another customization option, and your next task: replace the markers with images!

Images, Not Markers

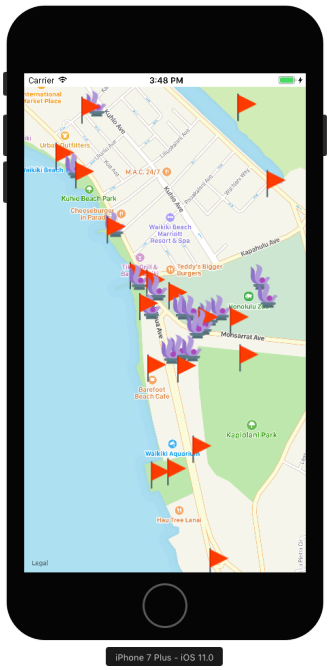
In *ArtworkViews.swift*, add the following class:

```
class ArtworkView: MKAnnotationView {
  override var annotation: MKAnnotation? {
    willSet {
      guard let artwork = newValue as? Artwork else {return}
      canShowCallout = true
      calloutOffset = CGPoint(x: -5, y: 5)
      rightCalloutAccessoryView = UIButton(type: .detailDisclosure)

      if let imageName = artwork.imageName {
        image = UIImage(named: imageName)
      } else {
        image = nil
      }
    }
  }
}
```

Now, you're using a plain old `MKAnnotationView` instead of an `MKMarkerAnnotationView`, and the view has an `image` property.
Back in *ViewController.swift*, in viewDidLoad(), register this new class, instead of `ArtworkMarkerView`:

```
mapView.register(ArtworkView.self,
  forAnnotationViewWithReuseIdentifier: MKMapViewDefaultAnnotationViewReuseIdentifier)
```

Build and run your app to see the sculptures and flags:



Now, you don't see the titles, but the map view shows *all* the annotations.

Custom Callout Accessory Views

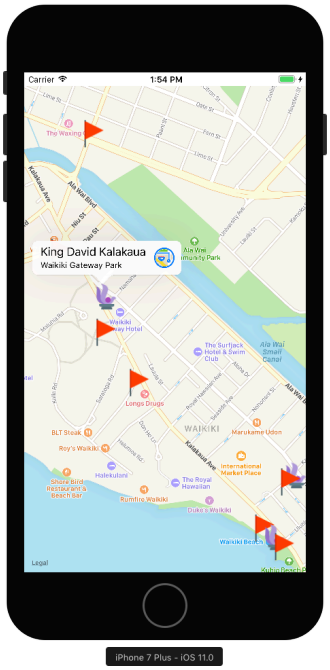
The right callout accessory is an info button, but tapping it opens the Maps app, so now you'll change the button to show the Maps icon.
Find this line in `ArtworkView`:

```
rightCalloutAccessoryView = UIButton(type: .detailDisclosure)
```

Replace this line with the following code:

```
let mapsButton = UIButton(frame: CGRect(origin: CGPoint.zero,
   size: CGSize(width: 30, height: 30)))
mapsButton.setBackgroundImage(UIImage(named: "Maps-icon"), for: UIControlState())
rightCalloutAccessoryView = mapsButton
```

Here, you create a `UIButton`, set its background image to the Maps icon from iconarchive.com in *Images.xcassets*, then set the view's right callout accessory to this button.
Build and run your app, then tap a view to see the new Maps button:



The final customization is the detail callout accessory: it's a single line, which is enough for the short location text, but what if you want to show a lot of text?
In *Artwork.swift*, locate this line in `init(json:)`:

```
self.locationName = json[12] as! String
```

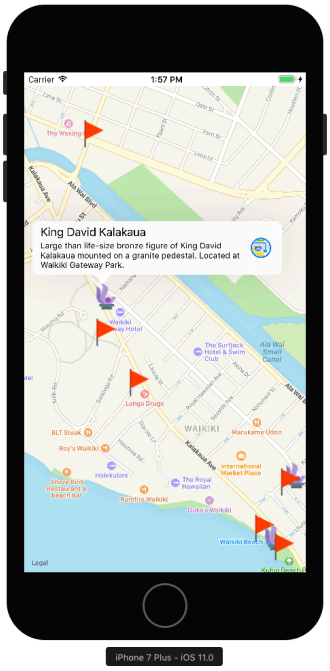
Replace it with this line:

```
self.locationName = json[11] as! String
```

Here you're opting for the *long* description of the artwork, which doesn't fit in the default one-line detail callout accessory. Now you need a multi-line label: add the following code to `ArtworkView`:

```
let detailLabel = UILabel()
detailLabel.numberOfLines = 0
detailLabel.font = detailLabel.font.withSize(12)
detailLabel.text = artwork.subtitle
detailCalloutAccessoryView = detailLabel
```

Build and run your app, then tap a view to see the long description:



Bonus Topic: User Location Authorization

This app doesn't need to ask the user for authorization to access their location, but it's something you might want to include in your other *MapKit*-based apps.

In *ViewController.swift*, add the following lines:

```
let locationManager = CLLocationManager()
func checkLocationAuthorizationStatus() {
  if CLLocationManager.authorizationStatus() == .authorizedWhenInUse {
    mapView.showsUserLocation = true
  } else {
    locationManager.requestWhenInUseAuthorization()
  }
}

override func viewDidAppear(_ animated: Bool) {
  super.viewDidAppear(animated)
  checkLocationAuthorizationStatus()
}
```

Here, you create a `CLLocationManager` to keep track of your app's authorization status for accessing the user's location. In `checkLocationAuthorizationStatus()`, you "tick" the map view's Shows-User-Location checkbox if your app is authorized; otherwise, you tell `locationManager` to request authorization from the user.

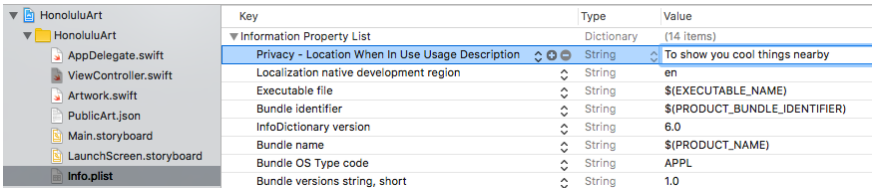
Note: The `locationManager` can make two kinds of authorization requests: `requestWhenInUseAuthorization` or `requestAlwaysAuthorization`. The first lets your app use location services *while it is in the foreground*; the second authorizes your app *whenever it is running*. Apple's documentation discourages the use of "Always":

Requesting "Always" authorization is discouraged because of the potential negative impacts to user privacy. You should request this level of authorization only when doing so offers a genuine benefit to the user.

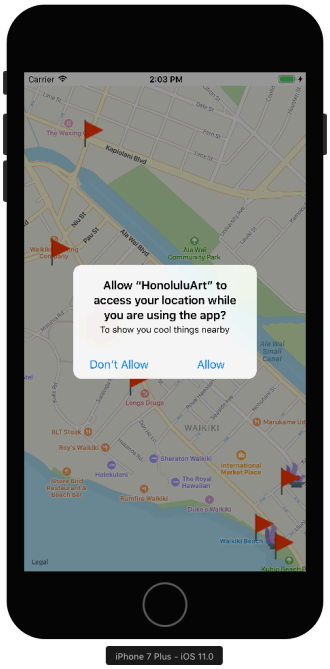
Info.plist item: important but easy to overlook!

There's just one more authorization-related task you need to do – if you don't, your app won't crash but the `locationManager`'s request won't appear. To get the request to work, you must provide a message explaining to the user why your app wants to access their location.

In *Info.plist*, open *Information Property List*. Hover your cursor over the up-down arrows, or click on any item in the list, to display the + and – symbols, then click the + symbol to create a new item. Scroll down to select *Privacy – Location When In Use Usage Description*, then set its *Value* to something like *To show you cool things nearby*:



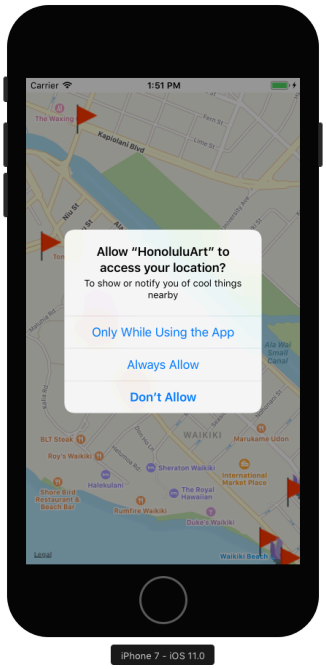
Build and run. You'll see the permission request appear on launch:



With a usage description like that, who wouldn't allow access? ;]

Note: As of iOS 11, you cannot request "Always" without offering "When in Use": if you set only *Privacy – Location Always Usage Description*, it won't appear, and you'll get the error message "Info.plist must contain both `NSLocationAlwaysAndWhenInUseUsageDescription` and `NSLocationWhenInUseUsageDescription` keys...". In Xcode 9 beta, I had to use the `NSLocationAlwaysAndWhenInUseUsageDescription` key; Xcode wouldn't select the matching Privacy key.

Below, the location manager requested "Always":



Where To Go From Here?

Here is the final project with all of the code you've developed in this tutorial.

Now you know the basics of using *MapKit*, but there's a lot more you can add: geocoding, geofencing, custom map overlays, and more. A great place to find additional information is Apple's Location and Maps Programming Guide.

Also look at WWDC 2017 Session 237 What's New in MapKit, to find more cool features they added in iOS 11.

We also have a terrific video course MapKit & Core Location that covers a lot of awesome topics.

Our 2-part How to Make an App Like RunKeeper tutorial shows you how to create your own run-tracker, using Core Location to map your run in real time.

If you want to decorate or customize the map provided by Apple with your own annotations and images, look at MapKit Tutorial: Overlay Views.

If you have any questions as you use MapKit in your apps, or tips for other MapKit users, please join in the forum discussion below!

Add a rating for this content

Thanks for your rating!