

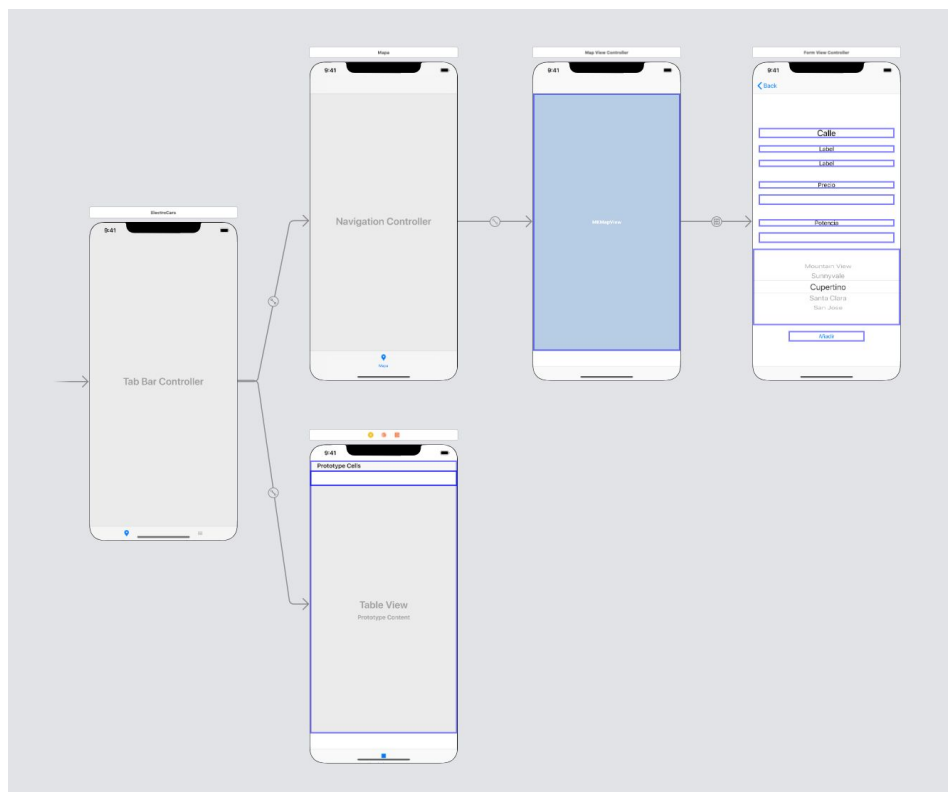
Tarea 1- iOS avanzado

MapKit y CoreLocation

Resumen

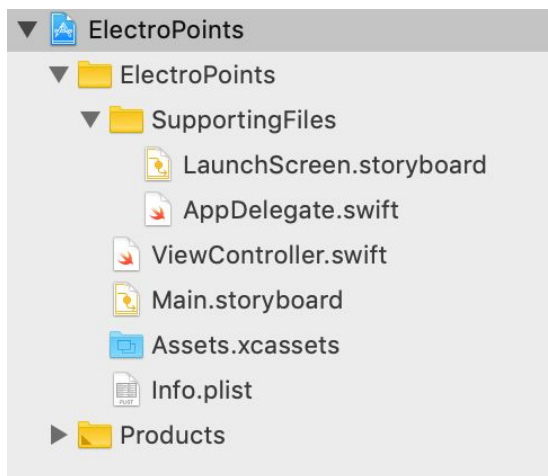
El objetivo de esta tarea es que el alumno se familiarice con dos de las librerías de iOS más utilizadas para el posicionamiento, las librerías [MapKit](#) y [CoreLocation](#), así como seguir reforzando conceptos relativos al diseño de interfaces con [UIKit](#) y persistencia de datos en [UserDefaults](#).

En esta tarea, el alumno desarrollará una app en la que se podrán dar de alta una serie de puntos de carga para coches eléctricos a través de una interfaz en MapKit, así como visualizar un listado de todos los puntos de carga registrados. El objetivo es reproducir la funcionalidad, de forma simplificada, de la app **Electromaps**.



Paso 0 - Creación del proyecto y organización de ficheros

1. Abrimos Xcode para crear un proyecto nuevo con la plantilla “*Single View App*”
2. Debemos darle un nombre al proyecto, en el caso de esta práctica, se ha llamado ***ElectroPoints***
3. Seleccionamos Swift como lenguaje y desactivamos las casillas de “*Core Data*”, “*Unit Tests*” y “*UI Tests*”
4. En el último paso, desactivaremos la casilla de “*Create Git repository on my Mac*”, ya que no vamos a utilizar un VCS en esta tarea
5. (Opcional) En la sección de la derecha de Xcode (Navigator) crearemos un nuevo grupo al que llamaremos Supporting Files, donde moveremos los ficheros “*AppDelegate.swift*” y *LaunchScreen.storyboard*, dado que no vamos a utilizarlos en esta tarea



Paso 1 - Creación de la vista del mapa MapViewController en un TabBar

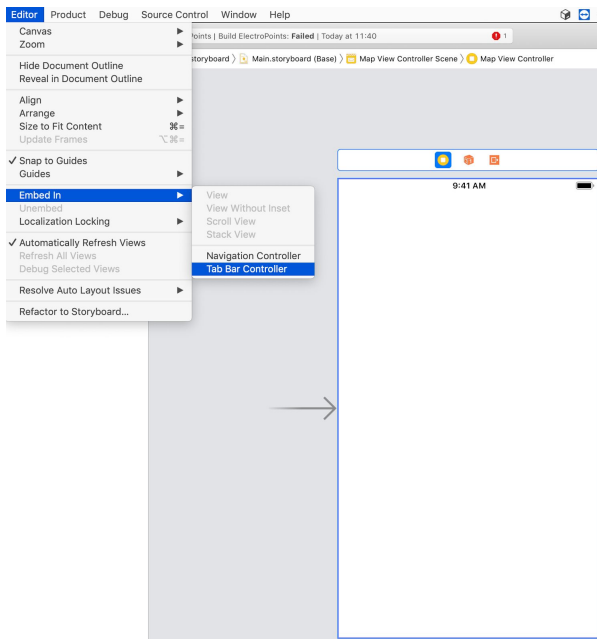
El primer paso será la creación de una vista para mostrar el mapa a través de MapKit.

Para ello, vamos a realizar los siguientes pasos:

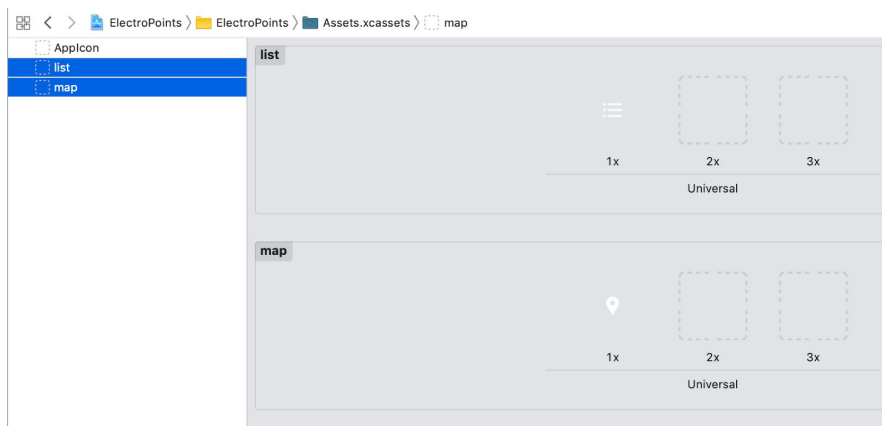
1. Dado que el proyecto genera una vista y un ViewController asociado a esta, renombrar el ViewController a MapViewController ($\text{⌘} + \text{click}$ sobre el nombre de la clase -> Rename)



2. Dado que en la app vamos a intercambiar entre la vista del mapa y el listado de puntos eléctricos a menudo, podemos embeber la vista del MapViewController en un TabBarController, en "Main.storyboard"

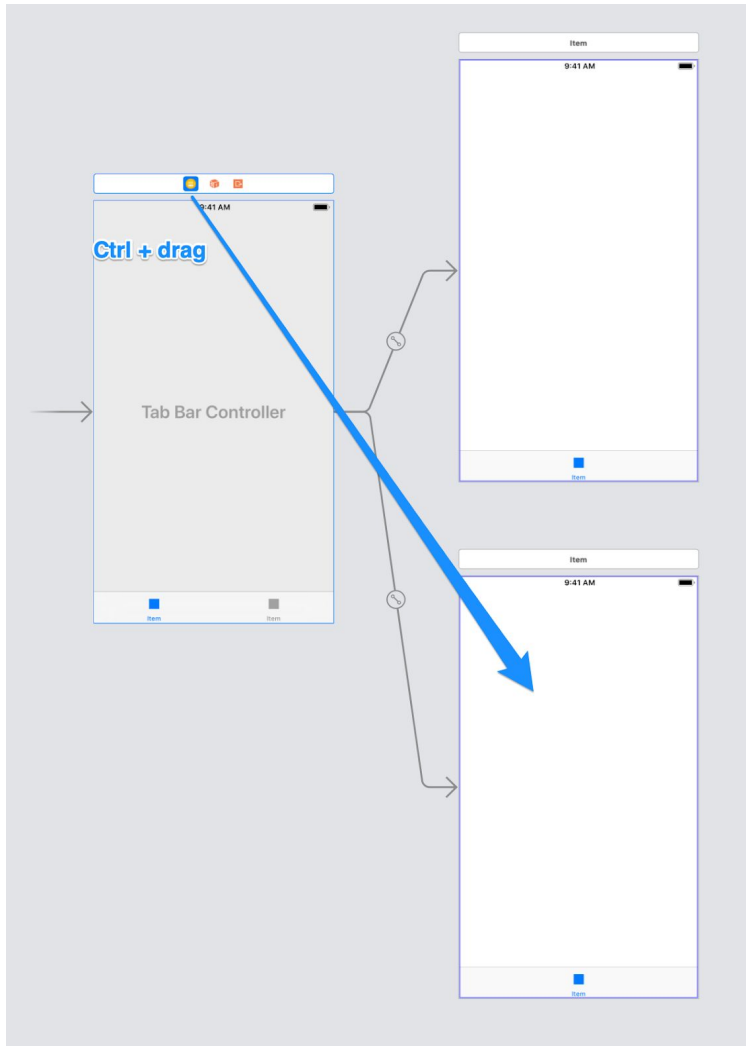


3. Un [TabBar](#) nos permite intercambiar entre diferentes mediante una barra en la parte inferior de la ventana. Dado que el objetivo es intercambiar entre 2 vistas (Mapa y Listado de puntos), se pide al alumno que añada al proyecto 2 imágenes de 25x25, una representando un mapa y otra un listado ([Custom Icons - Human Interface Guidelines](#))
- Webbs como [FlatIcon](#) permiten la descarga de este tipo de imágenes, acreditando a los autores
 - Para añadir una imagen al proyecto, hay que añadirla abriendo el fichero “Assets.xcassets” en Xcode, arrastrando tales imágenes sobre la barra lateral izquierda

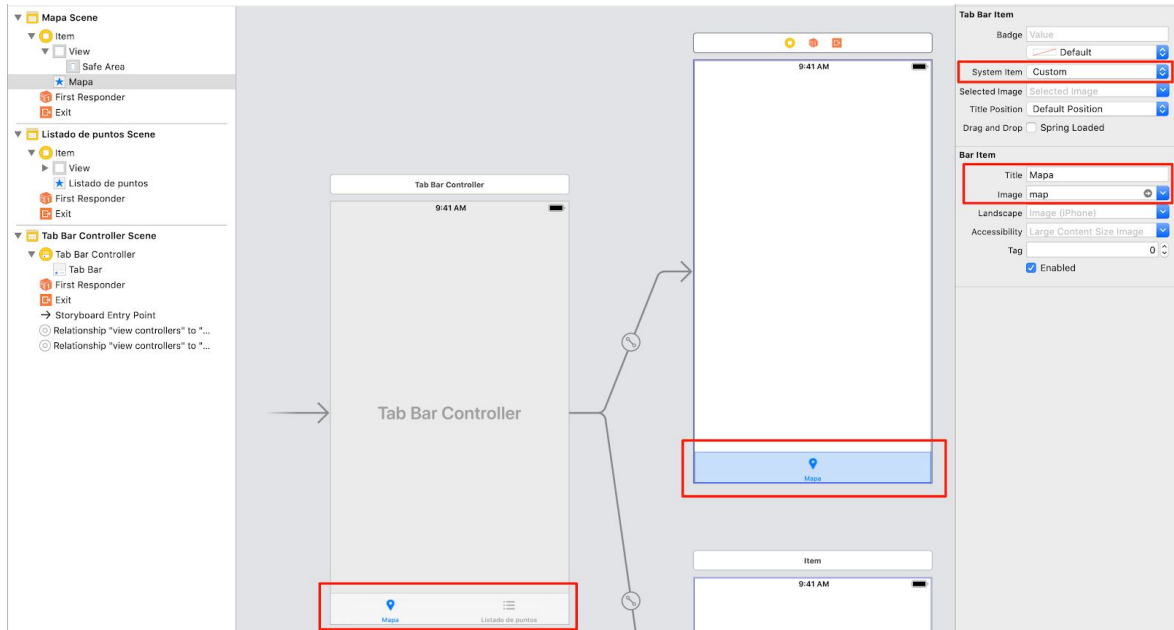


4. Dado que vamos a utilizar el TabBar para intercambiar entre dos vistas, el siguiente paso es crear otro ViewController en “Main.storyboard”
- Desde el object library, arrastramos un objeto de tipo View Controller

5. Para crear una relación entre el TabBarController principal y la nueva vista, deberemos hacer ctrl + drag desde el TabBarController hacia el nuevo ViewController, y establecer la conexión como *Relationship segue -> view controllers*



6. Solo queda modificar, en cada una de las vistas, la imagen del TabBarItem por las imágenes que hemos añadido



7. Si ejecutamos la app en este momento, podremos intercambiar entre las 2 vistas a través del TabBar

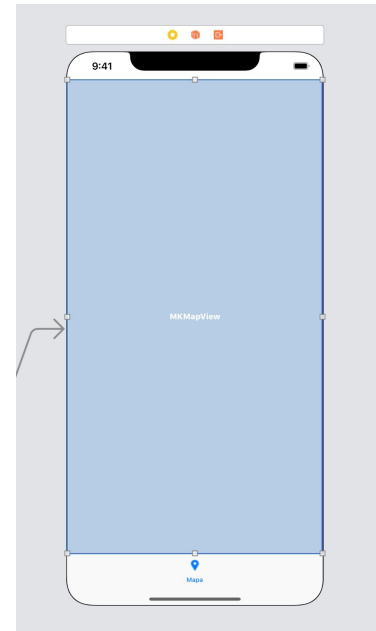
Paso 2 - Añadir MapKit y centrar una región inicial

Para que, al arrancar la app, aparezca un mapa centrado en una región en concreto, podemos seguir los siguientes pasos:

1. En "Main.storyboard", MapViewController, arrastramos un *Map Kit View*, y lo anclamos a los márgenes para que ocupe siempre todo el espacio disponible, respetando el TabBar inferior
2. En el **viewDidLoad()** de MapViewController, estableceremos la región inicial, para lo cual deberemos crear un @IBOutlet del mapa a través de la vista Assistant editor
 - a. IMPORTANTE: Si queremos trabajar con objetos de MapKit, deberemos importar la librería de MapKit en el fichero donde trabajemos con ella

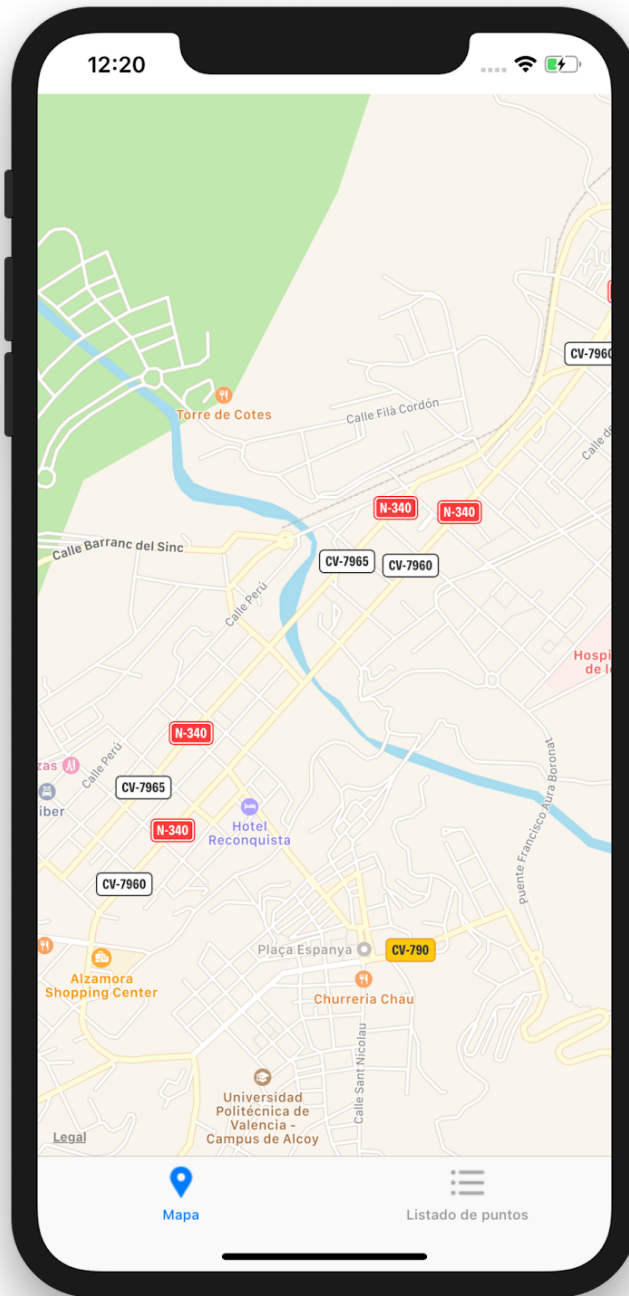
```
import UIKit
import MapKit

class MapViewController: UIViewController {
```



Una vez finalizados estos pasos, se pide al alumno que, en el **viewDidLoad()**

1. Instancie un objeto de tipo [CLLocation](#) a través del constructor **init(latitude: CLLocationDegrees, longitude: CLLocationDegrees)**
 - a. En este caso, se han elegido los valores 38.70545 y -0.47432 para centrar el mapa en la ciudad de Alcoy
2. Luego, instancie un objeto de tipo [MKCoordinateRegion](#) a través del constructor **init(center centerCoordinate: CLLocationCoordinate2D, latitudinalMeters: CLLocationDistance, longitudinalMeters: CLLocationDistance)**
 - a. Deberás utilizar la propiedad `coordinate` del objeto de tipo `CLLocation` para establecer el valor del parámetro `center`
3. Utilice el @IBOutlet del mapa para [establecer la región](#)



iPhone Xs Max - 12.1

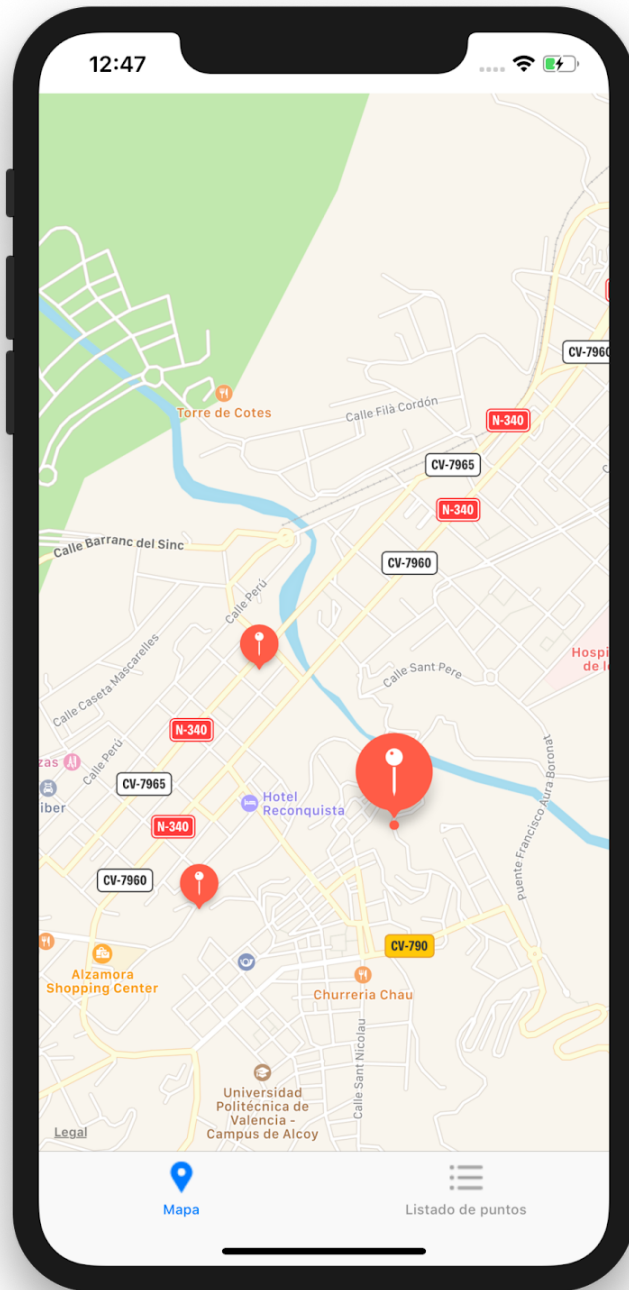
Paso 3 - Añadir una anotación al mapa con una pulsación larga

Para añadir puntos de carga a nuestra app, el usuario deberá hacer una pulsación larga en un punto del mapa, y por ello, el primer paso reside en detectar esta pulsación:

1. En el **viewDidLoad()**, después del código donde establecemos la región inicial, crearemos un objeto de tipo [UILongPressGestureRecognizer](#), y utilizar el [constructor init\(target: Any?, action: Selector?\)](#) para definir que la clase actual (self) será la encargada de actuar en caso de que esta pulsación larga ocurra, manejando tal caso a través de una función (selector)
2. [Añadir](#) al MapView ese Gesture recognizer
3. Definir la función que actúa como selector
 - a. Recuerda que la función que actúa como selector deberá estar anotada con @objc
 - b. Esta función aceptará un parámetro de tipo *UIGestureRecognizer*, que utilizaremos para filtrar los gestos detectados

Luego, en la función marcada como selector:

1. Comprobamos, a través del argumento de tipo *UIGestureRecognizer*, si el [estado del gesto](#) es [.began](#)
 - a. En tal caso, almacenamos en una variable de tipo *CGPoint* la [posición de tal gesto en la vista](#) del mapa
 - b. Luego, almacenamos en una variable de tipo [CLLocationCoordinate2D](#) el resultado de [convertir ese punto en sus correspondientes coordenadas de mapa](#) a través del objeto *MKMapView* (el @IBOutlet)
 - c. El siguiente paso es crear un objeto de tipo [MKPointAnnotation](#) y establecer su propiedad [coordinate](#) al objeto de tipo *CLLocationCoordinate2D*
 - d. El último paso es [añadir la anotación al mapa](#)



iPhone Xs Max - 12.1

Paso 4 - Extraer la dirección a través de geocodificación inversa

La geocodificación inversa es el paso de extraer, a través de unas coordenadas, una dirección física. En este caso, queremos conocer la calle en la que vamos a dar de alta el punto de carga.

1. Antes que nada, deberemos importar la librería CoreLocation en *"MapViewController.swift"*

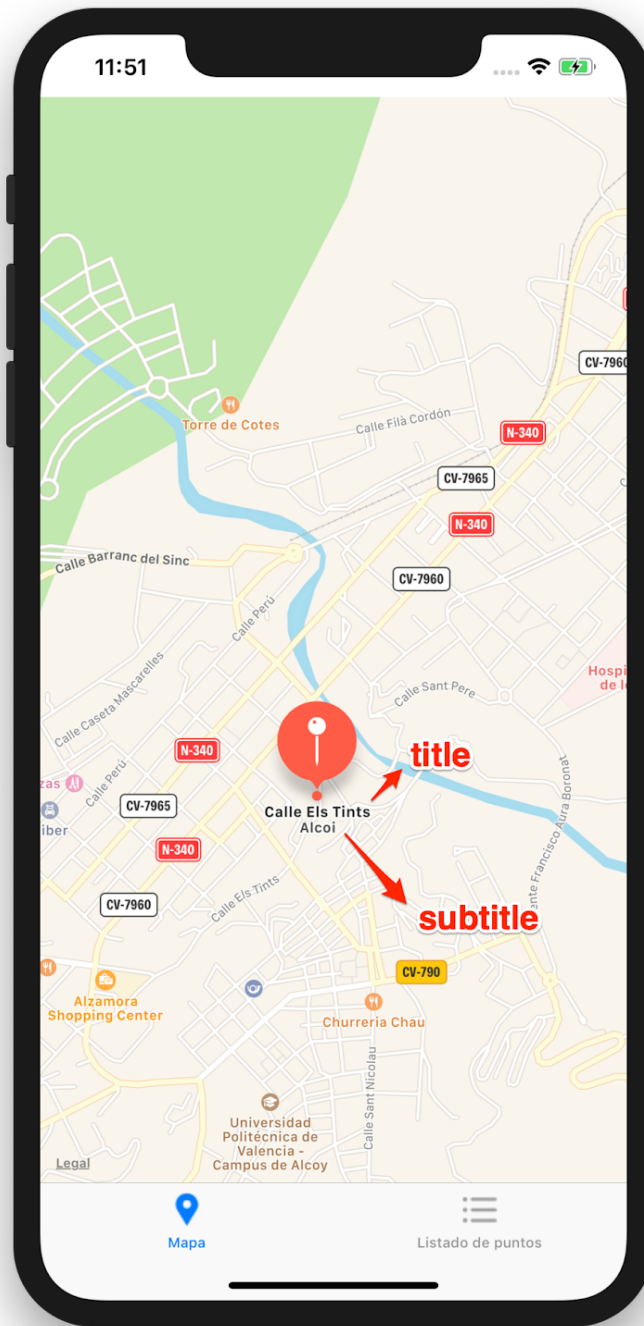
Para ello, vamos a modificar la función en la que añadimos un marcador al mapa (la función que detecta la pulsación larga)

1. En esa función, antes de añadir el marcador, crea un objeto de tipo [CLLocation](#) a partir de las coordenadas (tras convertir la posición del toque en pantalla a coordenadas)
2. Luego, crea un objeto de tipo [CLGeocoder](#) e invoca su función **reverseGeocodeLocation(_:completionHandler:)**, cuyo primer parámetro será el objeto CLLocation que acabas de crear
 - a. Esta función acepta una [completion handler](#) como segundo parámetro, que acepta dos parámetros, un [\[CLPlacemark\]?](#) y un [Error?](#)
 - i. Se adjunta una captura con la lambda, dado que la sintaxis puede parecer un poco extraña las primeras veces de su uso

```
// Reverse geocoding
let location = CLLocation(latitude: coords.latitude, longitude: coords.longitude)
CLGeocoder().reverseGeocodeLocation(location) { [unowned self] (placemarks, error) in

    // ...
}
```

3. En el completion handler, extrae el primer objeto del [\[CLPlacemark\]?](#)
4. Modifica las propiedades title y subtitle del objeto MKPointAnnotation para mostrar las propiedades [thoroughfare](#) y [locality](#) del objeto [CLPlacemark](#)
5. Mueve la línea que añade la anotación al mapa dentro del completion handler
 - a. Deberás utilizar la palabra self para hacer referencia al mapa dentro del handler
 - b. Si tienes dudas sobre el uso de la palabras unowned en este caso, puedes echar un vistazo a esta [entrada](#) del blog *krakendev*



iPhone Xs Max - 12.1

Paso 5 - Creación del modelo de datos

El siguiente paso consiste en crear una clase `ChargingPoint` para almacenar información sobre cada uno de los puntos de carga, por lo que deberás:

1. Crear un fichero Swift y declarar una clase `ChargingPoint` en ese fichero
2. Esa clase deberá heredar de `NSObject` y conformar el protocolo [MKAnnotation](#)
 - a. Deberás importar `MapKit` en el mismo fichero
 - b. Al implementar este protocolo, `MapKit` podrá mostrar objetos de este tipo en un mapa
3. Deberás declarar las siguientes propiedades de la clase
 - a. `name: String`
 - b. `street: String`
 - c. `power: Double`
 - d. `price: Double`
 - e. `type: ConnectorType`
4. Para conformar con el protocolo `MKAnnotation`, deberás [sobreescibir la propiedad](#) `coordinate`, de tipo `CLLocationCoordinate2D`
5. También es conveniente que sobreescribas las propiedades [title](#) y [subtitle](#) de `MKAnnotation` para, a través de una computed property, devolver las propiedades `name` y `street` respectivamente
6. Declara un enum `ConnectorType` con un raw value de tipo `Int`, con los valores 'schuko', 'mennekes' y 'chademo' (o los que quieras)
7. Finalmente, implementa el método constructor **`init()`** para inicializar todas las propiedades al crear una instancia según los parámetros del constructor
 - a. No te olvides de llamar al método **`super.init()`**
 - b. Deberás inicializar todos los campos, sin olvidar la propiedad `coordinate`

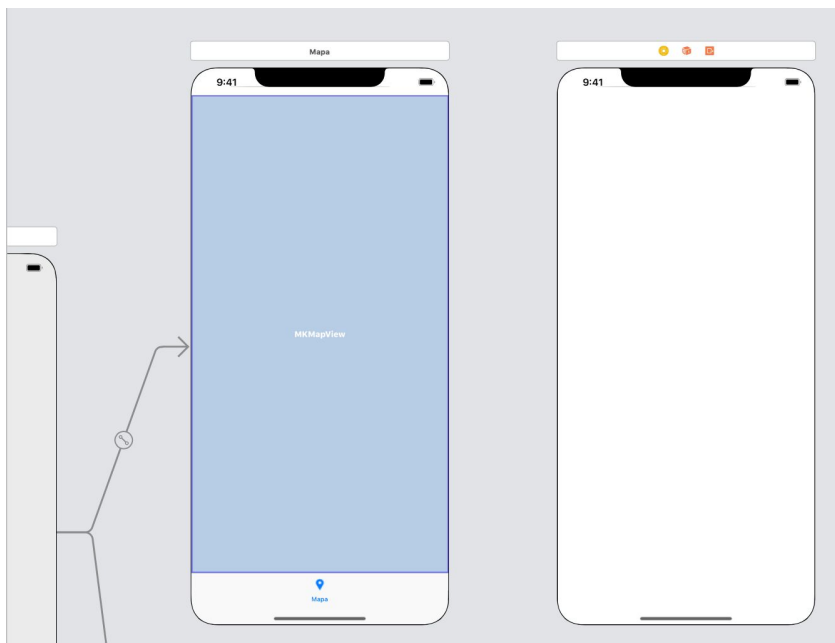
Ejercicio:

Dado que nuestra clase puede pintarse como una anotación por el hecho de conformar el protocolo `MKAnnotation`, se pide que el alumno reemplace, en el lugar donde se añade el marcador (detección de la pulsación larga), el objeto de tipo `MKAnnotation` por uno de tipo `ChargingPoint`, inicializando tal objeto con datos inventados, excepto la calle y la localidad.

Paso 6 - Creación de FormViewController para la inserción de datos adicionales

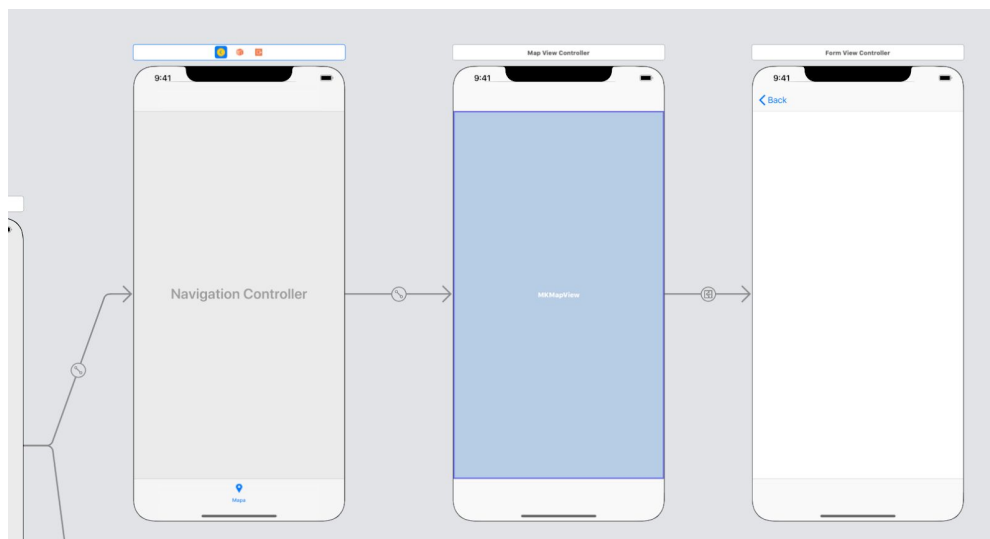
El siguiente paso es que el usuario pueda introducir datos como el precio, la potencia y el tipo de conector, ya que en el paso anterior los hemos escrito de forma literal. Para ello, deberás:

1. En "Main.storyboard", crea un ViewController, así como un fichero con la clase FormViewController: UIViewController, y asocia esa vista a tal clase
 - a. Dado que esta vista aparecerá tras pulsar en un lugar del mapa, podemos posicionar la vista al lado de MapViewController



2. Deberás crear un segue de tipo show, desde MapViewController hasta FormViewController, e identificarlo como "addPoint"
3. El siguiente paso, para comprobar que la navegación ocurre con éxito, será comentar todo el código del completion handler tras la realización de la geocodificación inversa y ejecutar el segue para navegar a FormViewController en su lugar
 - a. Deberás utilizar la palabra self para realizar el segue dentro del completion handler

Como verás, no hay forma de volver atrás, ni al mapa ni al listado (por ahora sin implementar) tras realizar el segue. Una solución fácil reside en embeber el `MapViewController` en un `Navigation Controller`




El siguiente paso a realizar es el paso de datos desde `MapViewController` a `FormViewController`, para lo que deberás seguir los siguientes pasos:

1. Declarar una propiedad de tipo `CLLocationCoordinate2D?` y otra de tipo `CLPlacemark?` en `FormViewController`
2. En el completion handler de `MapViewController`, al invocar el método **`performSegue(withIdentifier:sender:)`**, deberás pasar como sender una [tupla](#) con las coordenadas (`CLLocationCoordinate2D`) y el placemark (`CLPlacemark`)
3. Sobrecribir el método **`prepare(for:sender:)`** de `MapViewController`, en el que deberás seguir los siguientes pasos:
 - a. Convertir la propiedad [destination](#) del segue al tipo `FormViewController` (puedes utilizar guardas o un `if let`)
 - b. En caso de que la conversión sea satisfactoria, convertir el argumento pasado como sender al tipo (`CLLocationCoordinate2D`, `CLPlacemark`), es decir, recuperar la tupla, y almacenar el casting en una variable
 - c. Establecer las propiedades del `FormViewController` que has declarado en el punto 1 a partir de los valores de la tupla
4. Comprueba en el **`viewDidLoad()`** de `FormViewController` que el paso de datos ha sido satisfactorio

Paso 7 - Diseño del formulario y extracción de datos

En este paso, se pide que el alumno diseñe en *"Main.storyboard"*, el formulario de introducción de datos, que deberá:

1. Al aparecer la pantalla, mostrar en forma de texto la información obtenida a través del paso de datos desde MapViewController
 - a. thoroughfare (vía pública) a través del *CLPlacemark*
 - b. locality a través del *CLPlacemark*
 - c. coor a través del *CLLocationCoordinate2D*
2. Deberás permitir que el usuario introduzca, de una forma u otra, información sobre:
 - a. El precio, de tipo Double
 - b. La potencia, de tipo Double
 - c. El tipo de conector, de tipo ConnectorType
 - i. Se requiere el uso de un UIPickerView para visualizar las posibles opciones del enum, como ya se realizó en el curso anterior
 - ii. Se recomienda que el enum ConnectorType conforme con los protocolos [CaseIterable](#) y [CustomStringConvertible](#) para que sea más fácil trabajar con los casos
 - iii. Recuerda que deberás conformar los protocolos [UIPickerViewDelegate](#) y [UIPickerViewDataSource](#), a través de una extensión de la clase *FormViewController*, y asignar ambos a self (La clase FormViewController)
 1. Delegate:
 - a. **pickerView(_:titleForRow:forComponent:)** para definir qué texto aparece en cada fila del picker view
 2. DataSource:
 - a. **numberOfComponents(in:)** para saber cuántas columnas tiene el picker view
 - b. **pickerView(_:numberOfRowsInComponent:)** para saber cuántos elementos debe mostrar el picker view, según la columna
3. Finalmente, con un botón, deberás permitir confirmar la información para dar de alta el punto de carga, por lo que deberás generar un @IBAction



Una vez diseñada la interfaz del formulario, al detectar la pulsación sobre el botón para añadir un punto, deberás:

1. Realizar la extracción de los datos a través de los @IBOutlets para el precio, la potencia y el tipo de conector, y comprobar que los datos sean válidos
2. Instanciar un objeto de tipo ChargingPoint a partir de la información obtenida desde el formulario, y la recibida desde MapViewController a través de las propiedades de tipo *CLLocationCoordinate2D* y *CLPlacemark*

El último paso es devolver esa instancia de ChargingPoint a MapViewController para que pueda mostrarlo en el mapa, y para ello puedes seguir una de las siguientes estrategias:

1. O bien generar una propiedad en la clase MapViewController e ‘inflar’ ese dato desde FormViewController
2. O bien utilizar el patrón de delegates y protocolos para el paso de información entre las vistas (Recomendado)
 - a. En esta [entrada](#) de blog (*Passing Data Back With Properties And Functions (A ← B)*) se explican ambos casos, ya utilizados durante la realización de las prácticas del curso anterior

Finalmente, no te olvides de [desapilar](#) la vista de FormViewController para volver al mapa, a través del navigationController.

Deberás comprobar en este punto que, donde recoges la información del punto de carga en MapViewController, si añades esa anotación de tipo ChargingPoint al mapa, efectivamente se añade la anotación

1. Recuerda que esto es posible porque ChargingPoint conforma el protocolo MKAnnotation

Paso 8 - Almacenamiento de datos a través de un Singleton

Dado que la complejidad de esta app es relativamente simple y el acceso a los puntos de carga ha de hacerse desde varias vistas, el objetivo de este paso es el de la creación de un servicio de acceso a los datos, común a la app, a través del patrón de diseño [Singleton](#).

Para la creación del Singleton, necesitaremos crear un fichero Swift al que llamaremos “ChargingPointsService.swift”

1. Crea una clase PointsService
2. Declara una propiedad [static](#) llamada shared, e inicialízala a un objeto de tipo PointsService
3. Finalmente, declara una propiedad allPoints de tipo [ChargingPoints] e inicialízala a un array vacío
 - a. Es necesario inicializar ambas variables dado que no vamos a implementar el método init() para la clase
4. Deberás añadir cada punto de carga creado al array [ChargingPoints] a través de la propiedad estática shared
 - a. Se añade una captura sobre el uso del singleton para este caso en particular
 - b. En la captura, se ha optado por pasar los datos desde FormViewController a MapViewController utilizando protocolos y delegados

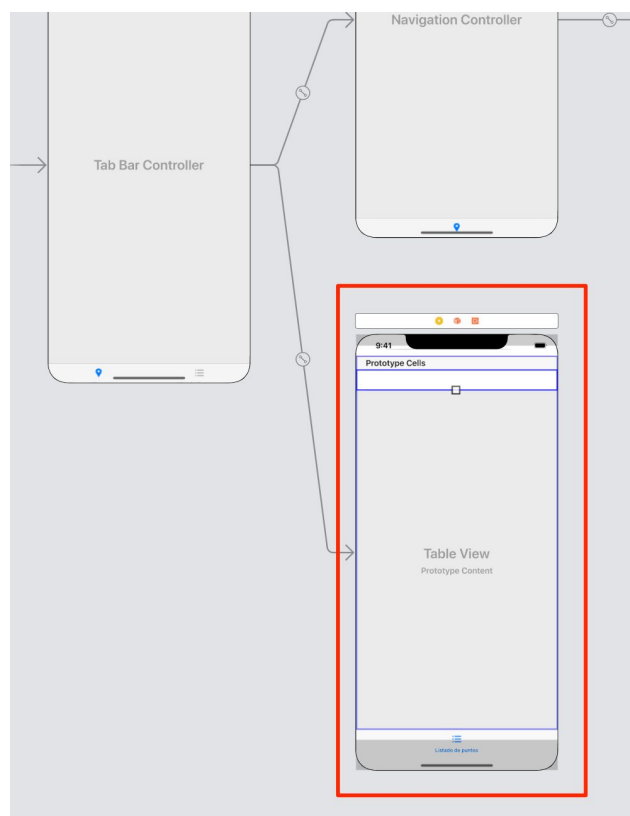
```
extension MapViewController: MapAnnotationDelegate {  
    func onChargingPointReady(point: ChargingPoint) {  
        mapView.addAnnotation(point)  
        PointsService.shared.allPoints.append(point)  
    }  
}
```

Paso 9 - Creación de ListViewController e implementación del TableView

En el siguiente paso, el objetivo es maquetar una vista, con su ViewController, en la que se muestre un TableView con tantos elementos como puntos de carga haya dado de alta el usuario.

Para ello, deberás:

1. Aprovechar la segunda vista ya creada en *"Main.storyboard"* (la que aparece al pulsar el TabBar) y asociarla a un fichero ViewController, como hemos estado haciendo hasta ahora, al que llamaremos ListViewController
2. En *"Main.storyboard"*, arrastrar un Table View en esa vista que acabas de crear
3. Arrastrar un Table View Cell a ese Table View, e identificarlo como "pointCell", por ejemplo

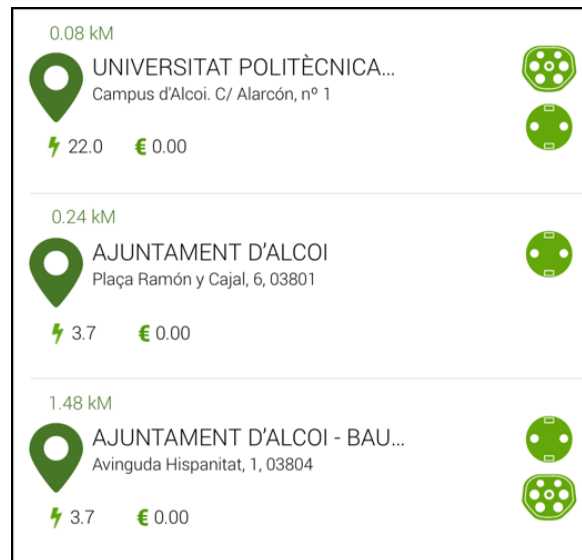


Ejercicio:

Se pide que el alumno customize una celda propia para la Table View.

Recuerda que deberás crear una clase que herede de [UITableViewCell](#), diseñar la celda desde “Main.storyboard” y conectar los @IBOutlets a la clase que acabas de crear.

Puedes inspirarte en el estilo de celdas utilizado por la app **Electromaps**.



El siguiente paso reside en ‘inflar’ cada una de las celdas con la información del punto de carga correspondiente, por lo que deberás:

1. Crear un @IBOutlet para la Table View
2. La clase ListViewController deberá conformar con el protocolo [UITableViewDataSource](#), a través de una extensión de la clase
3. Asignar, en la creación del Table View, que su dataSource será la clase ListViewController (self)
4. Implementar los siguientes métodos del dataSource
 - a. **tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int**
 - i. Devuelve el número de filas de la table
 - b. **tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell**
 - i. Devuelve la celda con la información

Ejercicio:

Tienes que implementar la función **tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell** para que devuelva una instancia de la celda customizada que has realizado en el ejercicio anterior.

Recuerda que en caso de no poder [desencolar](#) una celda con el identificador, puedes devolver una *UITableViewCell* nueva.

Finalmente, y dado que al utilizar el TabBar, la vista ListViewController solo se carga 1 vez, si queremos ver que aparece la información sobre los puntos de carga en la lista, deberás [detectar la aparición de la vista](#) y [recargar la información de la tabla](#).


Paso 10 - Adaptación de ChargingPoint para conformar el protocolo Codable

En el estado actual de la app, los puntos de carga que demos de alta son eliminados cuando el sistema libera nuestra app de la memoria del dispositivo. Por ello, el siguiente paso es hacer que esta información sea persistente entre diferentes ejecuciones, para lo que hay que seguir los siguientes pasos:

1. Modificar la clase ChargingPoint para que conforme con el protocolo [Codable](#)
 - i. Al escribir Codable al lado del nombre de la clase, Xcode nos dirá que tenemos errores en el código, dado que la clase no conforma con tal protocolo de por sí, por lo que debemos adaptar la clase
 - b. Debemos hacer una serie de modificaciones a la clase para que conforme con este protocolo
 - i. Declara un struct Coordinate que conforme con los protocolos [Codable](#) y [Hashable](#)
 - ii. Esta struct deberá tener 2 propiedades (latitude y longitude) de tipo Double
 - iii. La clase ChargingPoint deberá tener una propiedad de este tipo Coordinate, y llámala coordinates
 1. No puedes llamarla coordinate porque este nombre lo utiliza *MKAnnotation*
 2. Dado que el protocolo *MKAnnotation* no conforma con el protocolo *Codable*, debemos 'bypassear' este problema con un envoltorio en forma de computed property
 - iv. Convierte la propiedad coordinate de tipo *CLLocationCoordinate2D* a una computed property, devolviendo en su caso una nueva instancia de *CLLocationCoordinate2D* a partir de la propiedad de tipo Coordinate que acabas de crear

```
var coordinate: CLLocationCoordinate2D {  
    return CLLocationCoordinate2D(latitude: coordinates.latitude, longitude:  
        coordinates.longitude)  
}
```

- v. Deberás hacer que el enum ConnectorType también conforme con el protocolo Codable

- 
- vi. Finalmente, modifica el método **init()** para que acepte, en vez de un parámetro de tipo *CLLocationCoordinate2D* (que ahora es get-only), acepte uno de tipo *Coordinate*, e inicializa esa propiedad *coordinates*

Al haber modificado el método constructor, deberás adaptar, en *FormViewController*, el uso que le das a esa función. Al hacerlo, la app debería seguir ejecutándose como antes.

Paso 11 - Almacenamiento de datos en UserDefaults

Al igual que hicimos en la última tarea del curso anterior, vamos a almacenar los datos utilizando los [UserDefaults](#), para lo cual deberás realizar las siguientes adaptaciones en la clase `PointsService`:

Para guardar los datos en UserDefaults:

1. Añadir el property observer `didSet` al `[ChargingPoint]`
 - a. Utilizar una instancia de la clase [JSONEncoder](#) para [convertir](#) ese listado de puntos a JSON
 - i. Este paso es el mismo que el de la tarea 3 del curso anterior, por lo que puedes recuperar esa parte del código
 - b. Una vez convertido ese `[CharginPoint]` en un objeto de tipo [Data](#), almacenar ese objeto codificado en UserDefaults

Para cargar los datos desde UserDefaults:

1. Define una función **`loadPoints()`** en la misma clase (`PointsService`) cuyo objetivo será rellenar con el listado `[ChargingPoint]` con los datos almacenados en memoria
 - a. Accede al objeto compartido standard de la clase UserDefaults
 - b. Intenta hacer un casting a Data [accediendo al objeto almacenado](#) en UserDefaults
 - i. Si la conversión es satisfactoria, deberás intentar [decodificar](#), a través de una instancia de [JSONDecoder](#), el dato almacenado en UserDefaults, convirtiéndolo al tipo `[ChargingPoint]`
 1. Deberás utilizar la palabra `self` para establecer que el tipo destino de la conversión es `[ChargingPoint]`
 - ii. Finalmente, asigna a la propiedad donde almacenar todos los puntos de carga el resultado de la decodificación
 1. Todos estos pasos también son similares (con adaptaciones por los tipos específicos) a los seguidos en la tarea 3 del curso anterior

Paso 12 - Carga de los datos para mostrar los datos en el mapa

Ejercicio:

En este punto deberás averiguar cómo, al iniciar la app, cargar los puntos que has almacenado en memoria persistente a través de la función **loadPoints()** que implementaste en el punto anterior, y mostrarlos en el mapa.

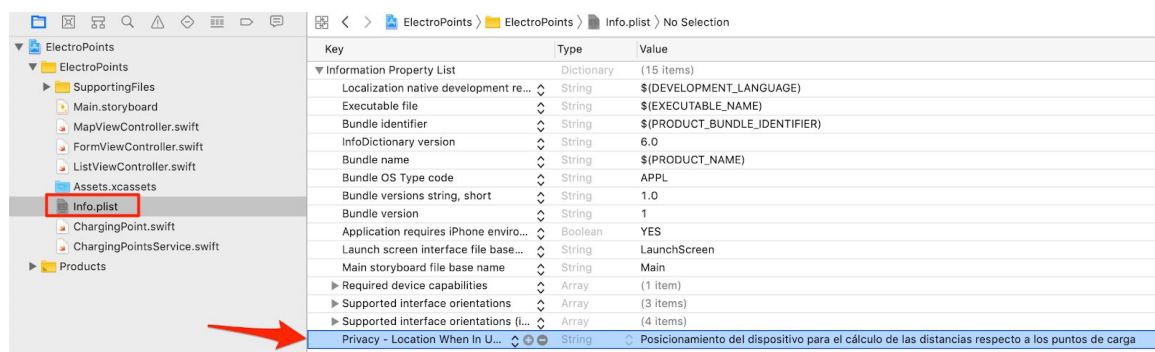
Puedes utilizar la función **addAnnotations(_:)** de *MKMapView* para [añadir un array](#) de `[ChargingPoint]` en vez de añadir cada punto uno a uno.

Paso 13 - Cálculo de distancia del usuario respecto a los puntos de carga

El último paso de esta tarea reside en calcular la distancia del usuario respecto a la de cada punto en particular. En este caso, el cálculo de la distancia deberá aparecer en cada una de las filas de la TableView de ListViewController.

Antes de empezar, y como debemos conocer cual es la posición del dispositivo a través del GPS, deberemos pedir los permisos adecuados:

1. Deberás añadir en el fichero *Info.plist* la clave “*Privacy - Location When In Use Usage Description*”, añadiendo una descripción sobre el uso que va a hacer la app del GPS



En iOS debemos formalizar desde código cuándo queremos pedir los permisos, detectar si el usuario ha concedido o no los permisos, y detectar posibles cambios en los permisos:

En ListViewController:

1. Importa la librería CoreLocation
2. Declara e inicializa una property de tipo [CLLocationManager](#)
3. En el **viewDidLoad()**, deberás asignar el delegate de [CLLocationManager](#) a self, y [pedir el permiso](#) 'whenInUse'
 - a. Seguidamente, en **viewDidLoad()**, [comprueba si el permiso](#) (método de la clase/tipo [CLLocationManager](#)) es 'authorizedWhenInUse', y, en caso que sea cierto, [solicita información sobre la posición](#) del dispositivo
4. Deberás hacer que la clase ListViewController conforme con el protocolo [CLLocationManagerDelegate](#) a través de una extensión, en la que deberás implementar los siguientes métodos:

- a. **locationManager(_ manager: CLLocationManager, didFailWithError error: Error)**
 - i. Para comprobar si ha habido algún error con la obtención de la posición
- b. **locationManager(_ manager: CLLocationManager, didChangeAuthorization status: CLAuthorizationStatus)**
 - i. En el momento en que el usuario cambie el estado del permiso, comprueba si la app tiene el permiso `authorizedWhenInUse` y solicita la información sobre la localización del dispositivo (similar al `viewDidLoad()`)
- c. **locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation])**
 - i. Esta función se ejecuta cuando el dispositivo recibe información sobre la posición del mismo (tras la solicitud) y cuando detecta cambios en la posición de este, y necesitamos esa información para calcular la distancia

Pasos para el cálculo de la distancia:

1. Declara en `ListViewController` una property `userLocation: CLLocation?`, que utilizaremos para almacenar la posición del usuario
2. En la función **`locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation])`** del `CLLocationManagerDelegate`, asignamos a esa property la última posición recibida por el delegate y recargamos la información de la Table View
3. Vamos a crear una función de instancia **`getDistance(from other: CLLocation) -> CLLocationDistance`** en la clase `ChargingPoint` para encapsular el cálculo de la distancia entre el punto actual y otro punto arbitrario
 - a. En esta función de instancia, genera un objeto de tipo `CLLocation` a partir de la property `coordinates` (de tipo `Coordinate`)
 - b. Utiliza la función **`distance(from location: CLLocation) -> CLLocationDistance`** de `CLLocation` para [devolver la distancia calculada](#) a partir de los dos objetos de tipo `CLLocation`
4. Finalmente, actualiza **`tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell`** de `ListViewController` para mostrar también la distancia, a partir del método auxiliar que acabas de definir

```
let distance = PointsService.shared.allPoints[indexPath.row]
    .getDistance(from: userLocation!)
```

Si quieres probar diferentes localizaciones en el simulador, podemos utilizar una serie de localizaciones específicas o utilizar una propia si introducimos las coordenadas:

