Mecolab - Nicolás Gebauer
22 de junio de 2015

Tutorial CoreData

Como usar una base de datos en Swift

Este tutorial busca enseñar como crear una nueva base de datos en Swift usando CoreData y como utilizarla. Abarca la creación de una nueva base de datos, la configuración de esta, el agregar nuevos objetos a esta, el obtener objetos de esta, el obtener objetos que cumplan ciertas condiciones de esta y el modificar y borrar estos objetos.

Puedes encontrar la versión actualizada de este tutorial aquí.

Tabla de contenidos

¿Por qué usar una base de datos? ¿Qué es?

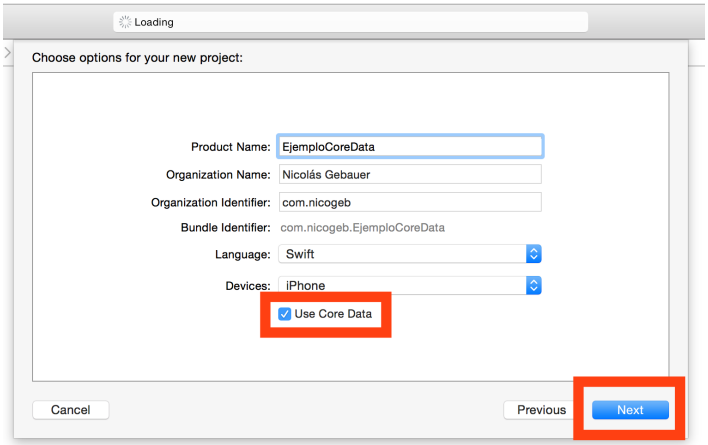
Una base de datos se usa cuando uno quiere guardar información que se mantenga a través de varias instancias de la aplicación, incluso cuando se cierre o se apague el dispositivo, a esto se le llama persistencia. Uno podría guardar todo en un archivo de texto con algún formato, como por ejemplo, "Nicolás;Gebauer;Martínez" para guardar un "Nombre;Apellido Paterno; Apellido Materno" o "2;300;4" para guardar "Nivel;Puntaje;Vidas" para guardar datos de un juego. Pero esto no es óptimo, además, es feo.

Para ello se crearon las bases de datos, donde la información se guarda de manera ordenada y se implementan métodos optimizados que permiten acceder a ella. En Swift, la base de datos por defecto es CoreData, que funciona con SQLite.

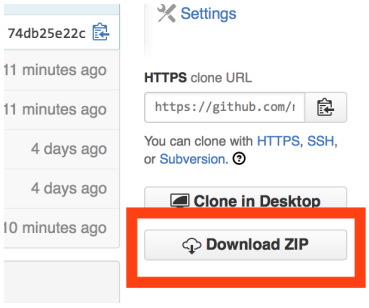
Empecemos con la parte importante ahora, ¿Cómo se usa?

Creando la base de datos

Para crear una base de datos lo primero es crear un nuevo proyecto. La diferencia esta en que al darle un nombre al proyecto debemos seleccionar la opción de usar Core Data como muestra la imagen.



Para este tutorial usaremos una interfaz gráfica que ya esta lista. Para ello, basta hacer clic aquí y descargar el proyecto haciendo clic en "Download ZIP". Extraemos el proyecto, abrimos la carpeta "EjemploCoreData-UI" y luego abrimos el archivo "EjemploCoreData.xcodeproj". Y renombramos el proyecto a "EjemploCoreData"



Notaras que en este proyecto hay archivos que no ves normalmente, como "EjemploCoreData.xcdatamodeld". Además, en la clase AppDelegate hay una nueva línea y dos nuevas funciones que serán muy importantes.

```
import CoreData

lazy var managedObjectContext: NSManagedObjectContext? = {
    let coordinator = self.persistentStoreCoordinator
    if coordinator == nil {
        return nil
    }
    var managedObjectContext = NSManagedObjectContext()
    managedObjectContext.persistentStoreCoordinator = coordinator
    return managedObjectContext
}()

func saveContext () {
    if let moc = self.managedObjectContext {
        var error: NSError? = nil
        if moc.hasChanges && !moc.save(&error) {
            NSLog("Unresolved error \(error), \(error!.userInfo)")
            abort()
        }
    }
}
```

La primera línea es la que tendremos que agregar a las clases donde manejemos la base de datos. La primera función la llamamos para obtener la referencia a la base de datos y la segunda es para guardar cambios en la base de datos. El hecho de que la variable sea lazy implica que el código para generar el valor que se retorna solo se corre una vez y el resultado queda guardado, así la variable no se regenera de nuevo cada vez que se accede a ella.

Una descripción breve de Core Data

Hagamos una pequeña pausa para saber un poco más de lo que vamos a usar. Core Data es un framework de object graph y persistencia creado por Apple para iOS y OSX. Core Data funciona bajo el modelo (schema) de la relación entre entidades y sus atributos para serializar objetos (en SQLite normalmente). Básicamente, Core Data permite manipular objetos, hacer tracking de las modificaciones a estos objetos y guardar los cambios en el disco pudiendo después obtener estos objetos del disco. Esto nos permite lograr persistencia.

CoreData logra esto a través de un *managed object context*. El context sirve como una puerta de entrada a una colección de objetos, conocida como el *persistence stack*, que hace la mediación entre los objetos en la aplicación y una base de datos.

Puedes leer más sobre CoreData aquí.

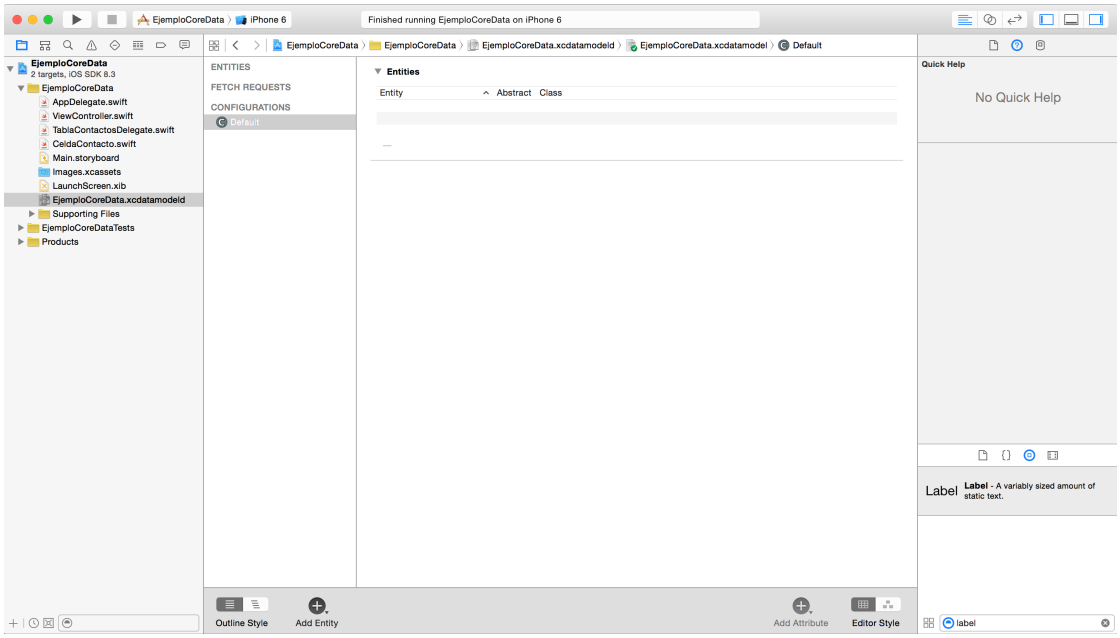
Como guardar información

En CoreData lo que se guardan son entidades, instancias de un objeto. Estas se guardan con sus atributos y pueden ser obtenidas de la base de datos después. Antes de poder crear,

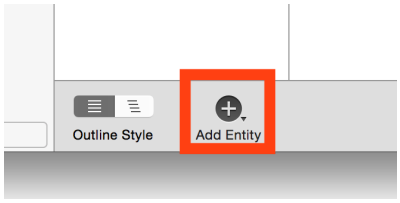
guardar y obtener entidades, debemos decirle a la base de datos que entidades guardará, que atributos tendrán y que relaciones habrá entre ellas.

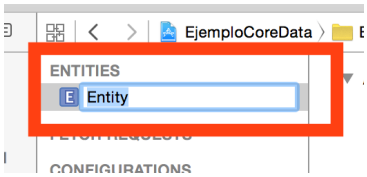
Definiendo las entidades

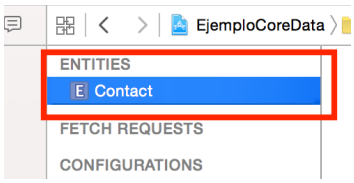
Para definir las entidades que usará la base de datos usamos el modelo (schema) "EjemploCoreData.xcdatamodeld". Al abrirlo deberías ver algo así:



Para este tutorial haremos una base de datos que guarde contactos. Primero, agregamos una nueva entidad haciendo clic en "Add Entity" y cambiamos su nombre a "Contact".

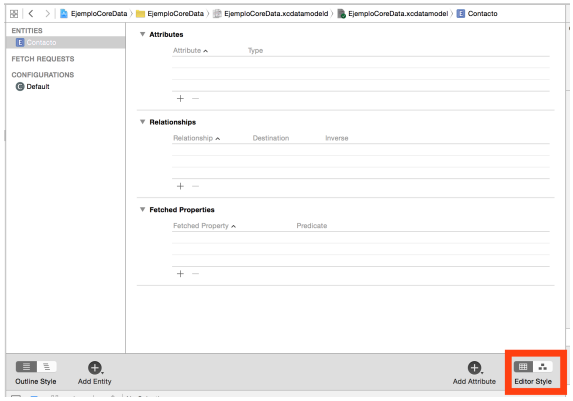




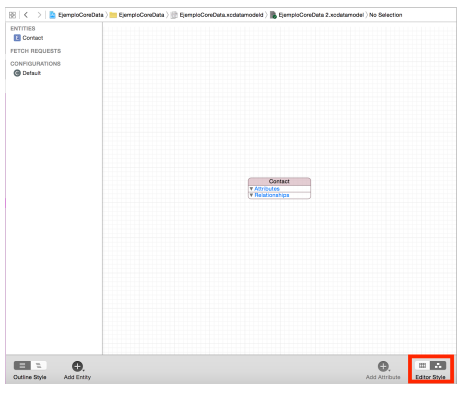


Nota: Existen dos estilos del editor, uno tipo tabla y uno gráfico, en este tutorial usaremos el editor de tipo tabla.

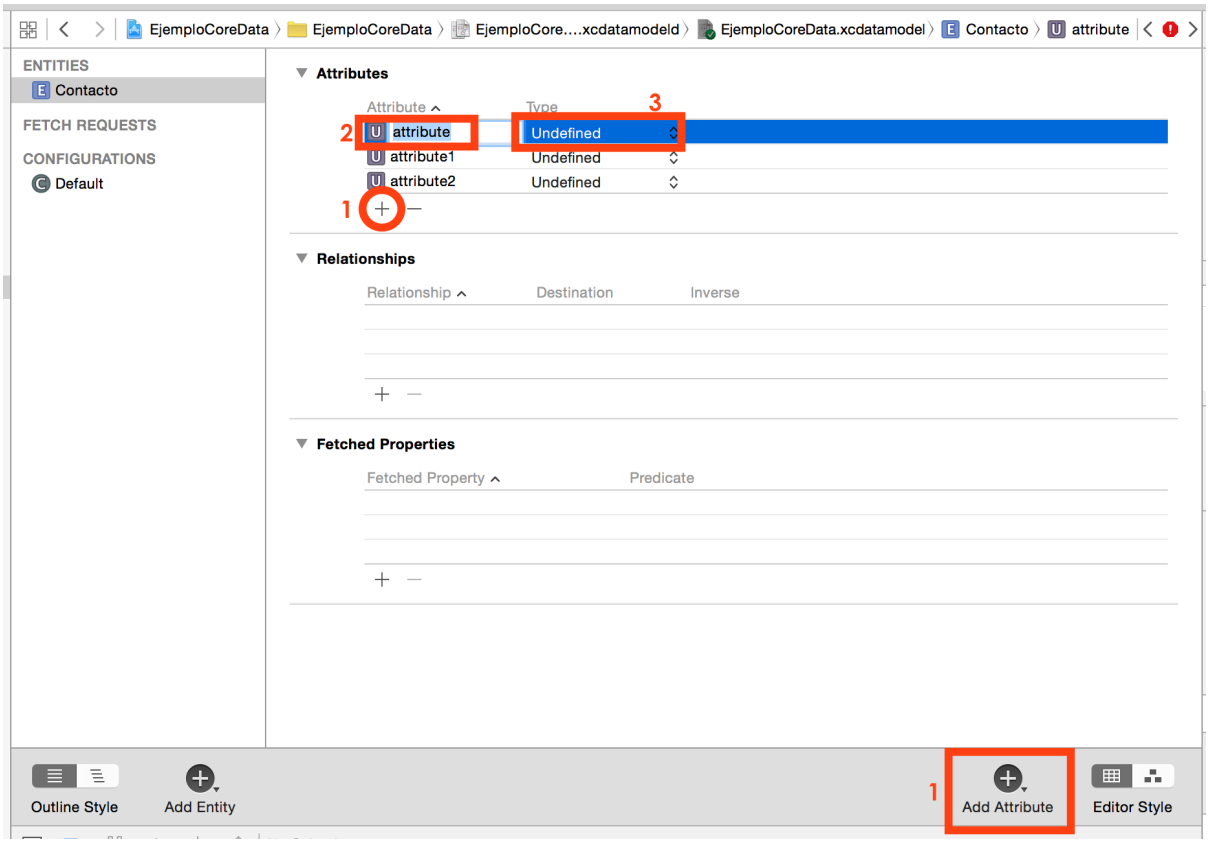
Editor tipo tabla

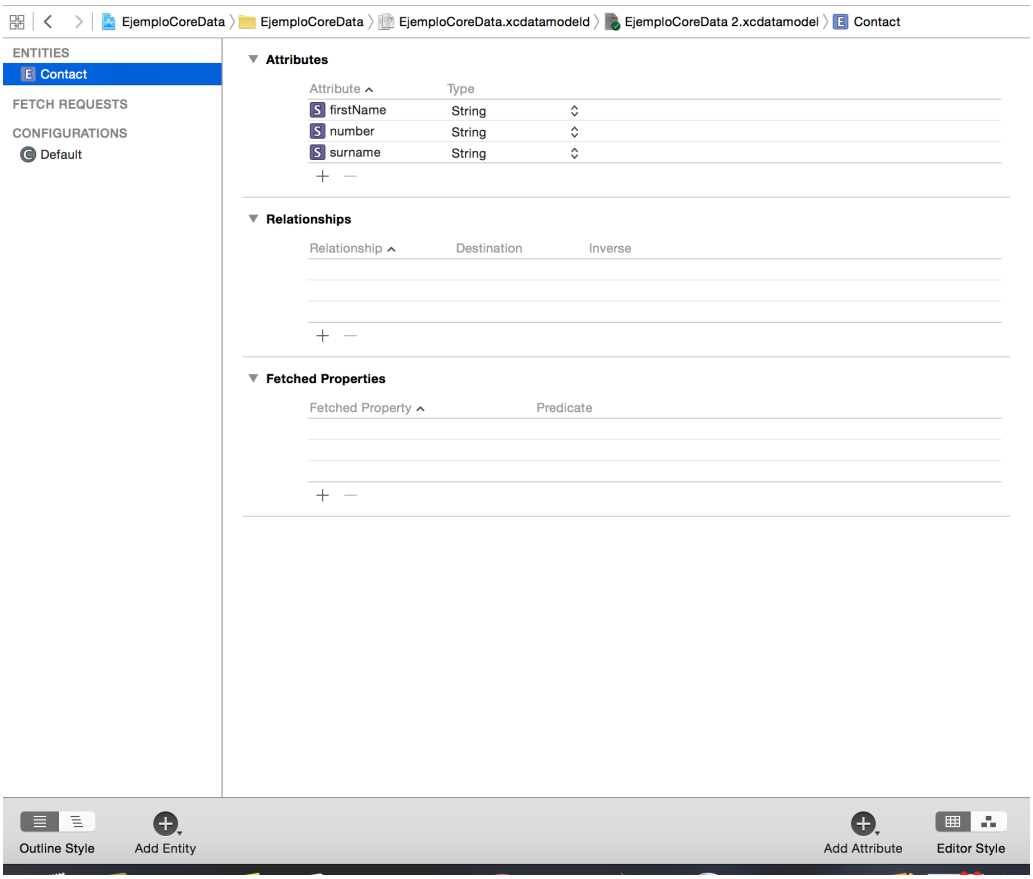


Editor gráfico



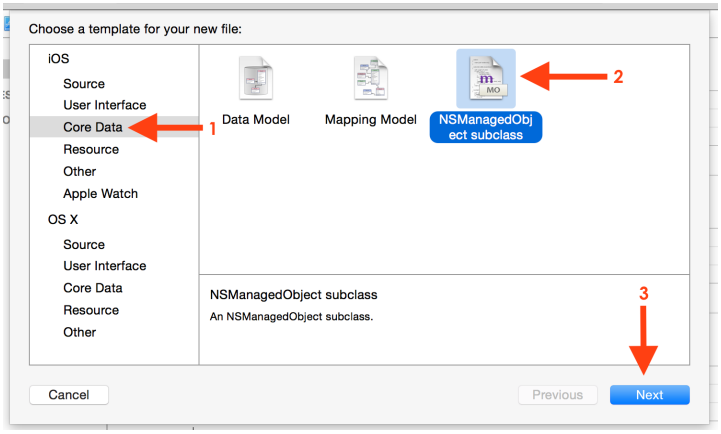
Nuestro *Contact* tendrá 3 atributos: Un *firstName*, un *surname* y un *number*. Para ello seleccionamos la entidad *Contact* y hacemos clic en "Add attribute" (**1**) 3 veces, luego modificamos el nombre de cada atributo (**2**) y le asignamos el tipo "*String*" (**3**).



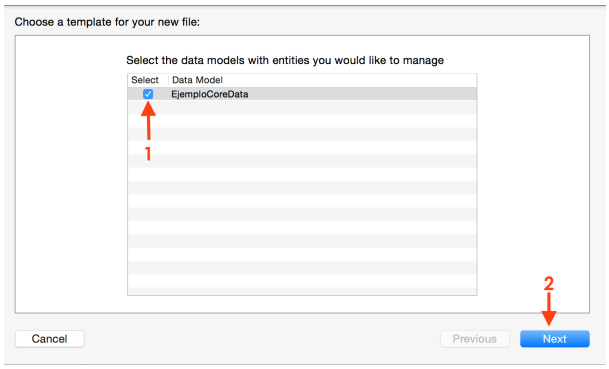


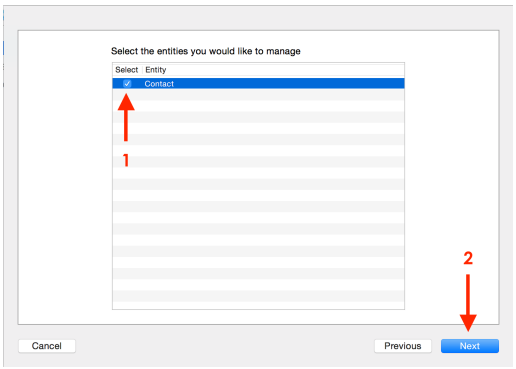
De esta manera, se pueden agregar atributos con otros tipos, siempre y cuando estos sean de tipos "básicos" como *String*, *Int*, *Bool*, etc. También se pueden guardar atributos de tipo *Binary Data* o *Transformable*, estos sirven para guardar cosas más complejas.

Para poder trabajar con las entidades guardadas en la base de datos o crear nuevas necesitamos una clase para cada entidad. Para crear esta clase de una manera rápida y automática creamos un nuevo archivo (Cmd + N), seleccionamos la opción "Core Data" (**1**), "NSManagedObject subclass" (**2**) y siguiente (**3**).

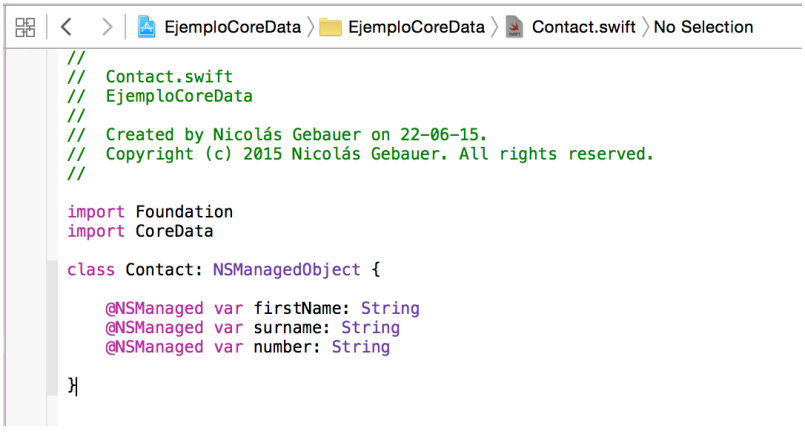


Luego nos aseguramos de que este seleccionado el modelo de base de datos que queremos manejar y la entidad que queremos manejar.



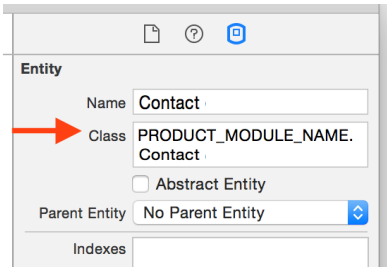


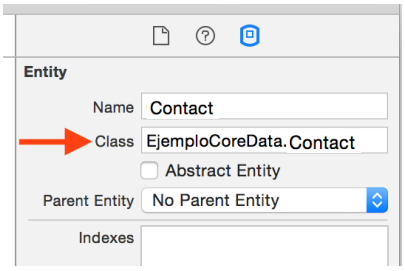
Y la clase *Contact* se habrá creado mágicamente.



Los atributos de esta clase tienen un nuevo prefijo *@NSManaged*, para saber qué hace esta propiedad prueba a borrarla. Veras que da un error de tipo "Class 'Contact' has no initializers" y "Stored property 'firstName' requires an initial value or should be *@NSManaged*". *@NSManaged* le dice al compilador, en palabras simples, que la inicialización de estas propiedades será manejada al correr el programa. En este caso, Core Data se hace cargo de esto, inicializando los atributos y modificándolos cuando interactuas con las entidades. Core Data se encarga de crear los métodos al ejecutar la aplicación.

Algo importante que se debe hacer antes de comenzar es la conexión entre la entidad del modelo y la clase que acabamos de crear. Para ello, vamos a nuestro modelo, hacemos clic en nuestra entidad *Contact* y en la barra lateral donde dice Class escribimos EjemploCoreData.Contacto





Y hasta aquí llega el trabajo "visual" y comienza el trabajo con código.

Preparándose para interactuar con la base de datos

Primero, crearemos una nueva clase que será la encargada de manejar la base de datos, la llamaremos "*ContactManager*". Recordemos que debemos importar CoreData y UIKit para obtener la referencia a la base de datos, por lo que la clase se vería así

```
import Foundation
import UIKit
import CoreData

class ContactManager {

}
```

Para poder manejar la base de datos necesitamos una referencia a esta, para ello agregamos un nuevo atributo a la clase *ContactManager* de la siguiente forma

```
let moc = (UIApplication.sharedApplication().delegate as!
AppDelegate).managedObjectContext
```

En este ejemplo, la vista principal interactúa con la base de datos, a través del botón de "Add contact" y "Fetch contacts", por lo que ella tendrá una instancia de *ContactManager* referenciada. También, le dará una referencia de esta a "*ContactsDelegateTable*" (el delegate) para poder cargar los datos de los contactos.

Agregamos el *contactManager* al *ViewController*:

```
let contacManager = ContactManager()
```

Y actualizamos la función viewDidLoad para realizar las conexiones:

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

        contactsTableDelegate = ContactsTableDelegate()
        contactsTableDelegate.refContactManager = contactsManager

        ContactsTable.delegate = contactsTableDelegate
        ContactsTable.dataSource = contactsTableDelegate
    }
```

Y al delegate le agregamos la referencia al *contactManager:*

```
weak var refContactManager: ContactManager!
```

Cuando hacemos clic en el botón "Add contact" queremos agregar un nuevo contacto a la base de datos. Implementemos esto

Agregar un nuevo contacto a la base de datos

Para agregar un nuevo contacto a la base de datos debemos crear una función en la clase que maneja este objeto que lo cree y lo guarde en la base de datos. Esta función recibirá un *firstName*, un *surname* y un *number* como argumentos. Para ello vamos a la clase *Contact* y agregamos la función que permitirá crear nuevos contactos.

```
class func new(moc: NSManagedObjectContext, firstName:String,
               surname:String, number:String) -> Contact {
    let newContact = NSEntityDescription.insertNewObjectForEntityForName("Contact",
               inManagedObjectContext: moc) as! EjemploCoreData.Contact

    newContact.firstName = firstName
    newContact.surname = surname
    newContact.number = number

    return newContact
}
```

Esta función creará una nueva instancia de *Contact*, la guardará en la base de datos y luego la retornará para poder usarla o modificarla.

Tenemos una clase que maneja los contactos (*ContactManager*) y una referencia a ella en el *ViewController* (*contactManager*), ahora tenemos que conectarlos.

En el *ContactManager* creamos una nueva función que nos permita agregar nuevos contactos a nuestra base de datos:

```
func addNewContact(firstName: String, surname: String, number: String) {
    _ = Contact.new(moc!, firstName: firstName, surname: surname, number: number)
}
```

Y la llamamos desde el *ViewController* en la acción *AddNewContact*:

```
@IBAction func addNewContact(sender: AnyObject) {
     contactsManager.addNewContact(TextFieldFirstName.text!,
       surname: TextFieldSurname.text!, number: TextFieldNumber.text!)

    deleteTextFields()
}
```

Y solo nos queda una cosa por hacer, grabar los cambios. Para ello creamos una nueva función en la clase *ContactManager*:

```
func saveDatabase() {
    do {
        try moc!.save()
    } catch {

    }
}
```

Esta función nos permite grabar los cambios en nuestra base de datos. ¿Por qué hay que hacerlo? Básicamente, cuando obtenemos objetos y los modificamos, agregamos, etc. trabajamos con copias de estos en la RAM, al ejecutar esta función nos aseguramos de que los cambios queden grabados en el disco duro.

Y agregamos una nueva línea en la función *addNewContact* para grabar la base de datos:

```
func addNewContact(firstName: String, surname: String, number: String) {
    _ = Contact.new(moc!, firstName: firstName, surname: surname, number: number)

    saveDatabase()
}
```

¡Felicidades! Puedes agregar nuevos contactos a tu base de datos y estos quedaran guardados aunque cierres la aplicación o apagues el teléfono. Ahora debemos mostrarlos.

Obtener los contactos de la base de datos

La clase *ContactManager* será la encargada de obtener los contactos y dárselos al delegate para que puedan ser mostrados en la tabla. Primero veamos como obtener todos los contactos. Para ello, agregamos un nuevo atributo en nuestra clase *ContactManager*:

```
var contacts = [Contact]()
```

Y creamos una nueva función llamada *fetchContacts* que nos permitirá obtenerlos de la base de datos:

```
func fetchContacts() {
    let fetchRequest = NSFetchRequest(entityName: "Contact")
    if let fetchResults = (try? moc!.executeFetchRequest(fetchRequest))
       as? [Contact] {
        contacts = fetchResults
    }
}
```

Ahora cada vez que hagamos *fetchContacts* nuestro array de contactos se actualizará con todos los contactos de la base de datos. Queremos que los contactos se muestren desde que se inicia la aplicación, así que crearemos el método init y llamaremos a *fetchContacts*

```
init() {
    fetchContacts()
}
```

Solo nos falta mostrar esta información en nuestra tabla, configurando nuestro delegate:

```
func tableView(tableView: UITableView,
       cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cellContact = tableView.dequeueReusableCellWithIdentifier(IDContactCell)
       as! ContactCell

    let contact = refContactManager.contacts[indexPath.row]

    cellContact.LabelNumber.text = contact.number
    cellContact.LabelName.text = contact.firstName + " " + (contact.surname)

    return cellContact
}

func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return refContactManager.contacts.count
}
```

Ahora solo falta que al hacer clic en “Fetch contacts” y que estos se actualicen, para ello modificamos nuestra función *fetchContacts* en nuestro *ViewController* para que le diga al *contactManager* que haga el fetch:

```
@IBAction func fetchContacts(sender: AnyObject) {
    contactsManager.fetchContacts()
    ContactsTable.reloadData()
}
```

Adelante, prueba la aplicación. Agrega nuevos contactos, cierra la aplicación y ve como se vuelven a cargar al abrirla. Haz clic en "Fetch contacts" para actualizar la lista luego de agregar nuevos contactos. ¡Ya sabes lo básico de Core Data!

Como borrar objetos de la base de datos

Supongamos que ahora queremos borrar un contacto de nuestra base de datos, esto se puede hacer fácilmente en la aplicación que tenemos. Para eliminar un objeto de la base de datos se utiliza el comando:

```
deleteObject(object: NSManagedObject)
```

Para utilizarlo, crearemos una nueva función en la clase *ContactManager* que recibirá un index y eliminara el contacto correspondiente a ese índex en el array de contactos:

```
func deleteCar(index:Int) {
   if index < cars.count {
        moc?.deleteObject(cars[index])
   }
   saveDatabase()
}
```

Para llamarla desde nuestro delegate debemos permitir la edición de las celdas y que el tipo de la edición sea la de eliminar, llamando a la función que acabamos de crear:

```
func tableView(tableView: UITableView, canEditRowAtIndexPath indexPath:
NSIndexPath) -> Bool {
    return true
}

func tableView(tableView: UITableView,
    commitEditingStyle editingStyle: UITableViewCellEditingStyle,
    forRowAtIndexPath indexPath: NSIndexPath) {
        if (editingStyle == .Delete) {
            refContactManager.deleteContact(indexPath.row)
            refContactManager.fetchContacts()
            refContactsTable.deleteRowsAtIndexPaths([indexPath],
withRowAnimation: .Automatic)
        }
        tableView.setEditing(false, animated: true)
}
```

Recuerda que debes hacer clic en "Fetch contacts" para que se actualice la tabla ya que esto no se hace automáticamente. Prueba a agregar un contacto y luego borrarlo.

Búsqueda avanzada de objetos (fetch con predicate)

Qué pasa si solo queremos buscar los contactos que cumplan con una cierta condición, como por ejemplo, que su nombre sea Juan, o que su numero contenga 569, ¿Revisamos todos los contactos con un for y luego vemos cuales cumples esta condición? Afortunadamente, no. Core Data nos proporciona una manera de hacer búsquedas con filtros predefinidos y también hacer que estas búsquedas se ordenen (por ejemplo, una lista de contactos por apellido). Primero veamos como indicarle a la base de datos que ordene los objetos que retorne y luego como filtrar el resultado deseado.

Ordenando los resultados del fetch

Para que los resultados de un fetch estén ordenados basta que nuestra función fetchContactos se modifique para tener un descriptor de orden que tome en cuenta el *firstName* de cada contacto para saber cual va primero:

```
func fetchContacts() {
    let sortDescriptor = NSSortDescriptor(key: "firstName", ascending: true)
    let fetchRequest = NSFetchRequest(entityName: "Contact")
    fetchRequest.sortDescriptors = [sortDescriptor]

    if let fetchResults = (try? moc!.executeFetchRequest(fetchRequest))
            as? [Contact] {
        contacts = fetchResults
    }
}
```

sortDescriptor le dice a nuestro *fetchRequest* como debe ordenar los resultados. En este caso utiliza el parámetro *firstName* para ordenar del más pequeño al más grande, esto ya que ascending = true, por lo que los contactos se ordenaran por abecedario usando el *firstName*.

Filtrando los resultados del fetch

Para que un fetch nos retorne solamente objetos que cumplan con una cierta condición creamos un predicate. Por ejemplo, si queremos un fetch que retorne solo contactos cuyo nombre contenga “co”:

```
let predicateFirstName = NSPredicate(format: "firstName CONTAINS %@", "co")
let fetchRequest = NSFetchRequest(entityName: "Contact")
fetchRequest.predicate = predicateFirstName
if let fetchResults = moc!.executeFetchRequest(fetchRequest, error: nil)
     as? [Contact] {
    contacts = fetchResults
}
```

En este caso contactos sería un array con los contactos cuyo *firstName* contiene “co”.

Además, si queremos hacer una búsqueda con varios predicates podemos crearlos y luego juntarlos en uno solo:

```
let f = NSPredicate(format: "firstName CONTAINS %@", "co")
let s = NSPredicate(format: "surname CONTAINS %@", "ge")
let p = NSCompoundPredicate(type: NSCompoundPredicateType.AndPredicateType,
     subpredicates: [f,s])
let fetchRequest = NSFetchRequest(entityName: "Contact")
fetchRequest.predicate = p
if let fetchResults = moc!.executeFetchRequest(fetchRequest, error: nil)
     as? [Contact] {
    contacts = fetchResults
}
```

En este caso contactos sería un array con los contactos cuyo nombre *firstName* "co" y cuyo *surname* contiene "ge", lo cual esta dado por el tipo de *NSCompoundPredicateType*. También podría haber sido de tipo *or* con lo que contactos sería un array con los contactos cuyo *firstName* contiene "co" o cuyo *surname* contiene "ge". Así se pueden formar predicates más complejos combinando predicates simples. Incluso se puede combinar un *NSCompoundPredicate* con otro para formar un predicate muy complejo.

Para este tutorial, implementaremos los filtros utilizando los text field ya disponibles. Solo revisaremos para casos donde se cumplan todos los filtros (AndPredicateType). Para ello, vamos a agregar una nueva función a nuestro *ContactManager* que recibirá argumentos para ejecutar su fetch con filtros:

```
func fetchContactsWithPredicates(firstName:String, surname:String, number:String) {
    var predicatesArray = [NSPredicate]()

    if firstName != "" {
        let predicateFirstName = NSPredicate(format: "firstName CONTAINS %@",
             firstName)
        predicatesArray.append(predicateFirstName)
    }
    if surname != "" {
        let predicateSurname = NSPredicate(format: "surname CONTAINS %@", surname)
        predicatesArray.append(predicateSurname)
    }
    if number != "" {
        let predicateNumber = NSPredicate(format: "number CONTAINS %@", number)
        predicatesArray.append(predicateNumber)
    }
    let predicate = NSCompoundPredicate(type:
       NSCompoundPredicateType.AndPredicateType, subpredicates: predicatesArray)
    let sortDescriptor = NSSortDescriptor(key: "firstName", ascending: true)
    let fetchRequest = NSFetchRequest(entityName: "Contact")
    fetchRequest.sortDescriptors = [sortDescriptor]
    fetchRequest.predicate = predicate

    if let fetchResults = (try? moc!.executeFetchRequest(fetchRequest))
       as? [Contact] {
        contacts = fetchResults
    }
}
```

Y modificaremos la función *FetchContacts* del *ViewController* para reaccionar cuando haya filtros que aplicar:

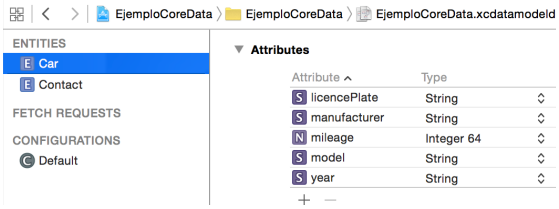
```
@IBAction func fetchContacts(sender: AnyObject) {
    if (TextFieldFirstName.text != "" ||
        TextFieldSurname.text != "" ||
        TextFieldNumber.text != "") {
            contactsManager.fetchContactsWithPredicates(
                TextFieldFirstName.text!,
                surname: TextFieldSurname.text!,
                number: TextFieldNumber.text!)
    } else {
        contactsManager.fetchContacts()
    }
    deleteTextFields()

    ContactsTable.reloadData()
}
```

Ahora si escribes algo en los text fields, esta información se usará para filtrar tu fetch. Compruébalo escribiendo distintos términos y haciendo clic en “Fetch contacts".

Entidades cuyos atributos son otras entidades: Relaciones

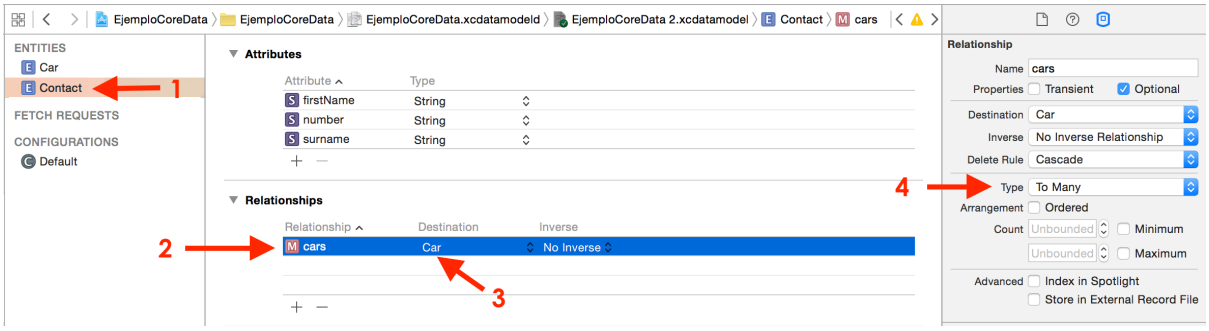
Puede darse el caso que una de las entidades que guardemos tenga una referencia a una o varias entidades. Como te podrás haber dado cuenta, los atributos que ofrece el editor de la base de datos son limitados. Para poder lograr lo que queremos usaremos relaciones.

Las relaciones se refieren a entidades que se relacionan con otras entidades, ya sea que una entidad contiene muchas entidades o que dos entidades están estrechamente relacionadas. Imaginemos que cada *Contact* tiene un *car* y cada *car* tiene sus propias características como *manufacturer*, *year*, *mileage*, etc. En vez de que cada *Contact* guarde todos estos atributos, cada *Contact* podrá tener uno más cars, los cuales estarán definidos por su propia entidad: *Car*. Comenzamos creando nuestra nueva entidad *Car*, vamos al modelo de nuestra base de datos y creamos la nueva entidad, agregamos las propiedades *manufacturer*, *year* y *model*, todas de tipo String y *mileage* de tipo Integer 64. Nuestra base de datos debería quedar como la de la imagen.

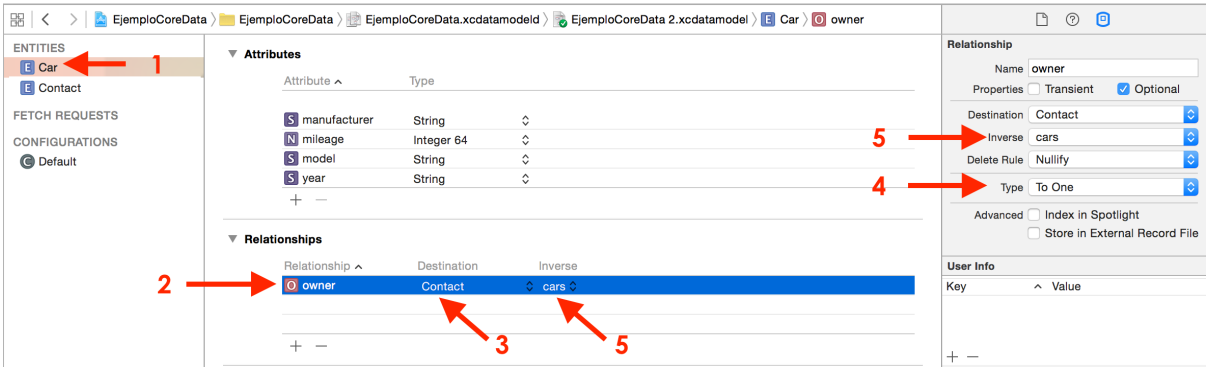


Ahora, cada *Contact* podría tener uno o más cars, así que vamos a la entidad *Contact* (**1**), agregamos una nueva relación que se llamara

"*cars*" (**2**), su destino será la entidad *Car* (**3**) y su relación será de tipo To Many (**3**).

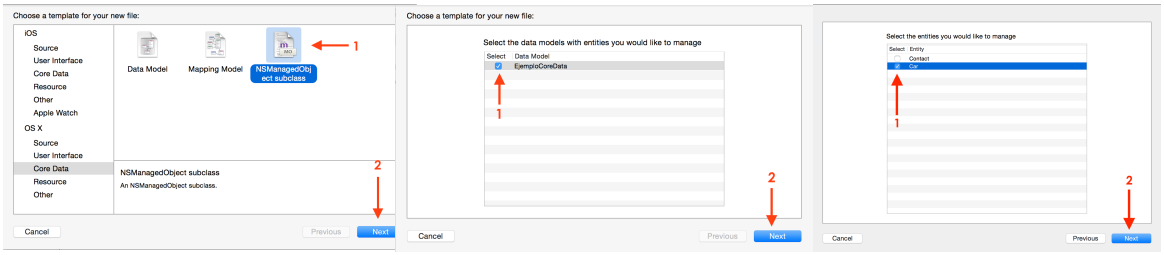


Luego debemos hacer la relación inversa, es decir, del *Car* al *Contact*. Cada *Car* tendrá un solo *owner*, por lo que vamos a la entidad *Car* (**1**), agregamos una nueva relación que se llamara "*owner*" (**2**), su destino será la entidad *Contact* (**3**) y su relación será de tipo To One (**4**), además, su inversa será la relación "*cars*" (**5**).

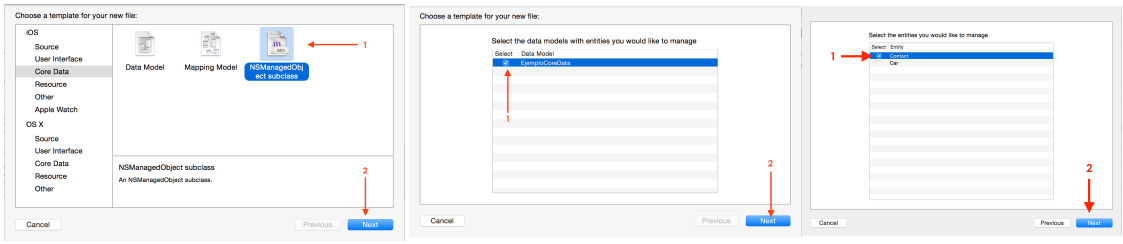


Ahora cuando volvemos a la entidad *Contact* te darás cuenta que la relación inversa se agregó automáticamente. Esto significa que cada *Contact* puede tener uno o más *cars*, cada uno con sus propias propiedades, pero cada *Car* solo puede tener un *owner* (*Contact*).

Ahora creemos la nueva clase *Car* y actualicemos nuestra clase *Contact*. Creamos una nueva clase de tipo NSManagedObject subclass para la entidad *Car*.

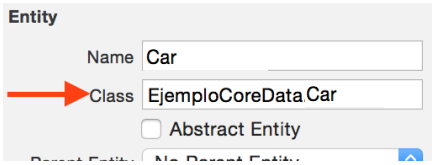


Ahora actualizamos la clase *Contact*, para ello copiamos la función *new* y volvemos a crear la clase al igual que la primera vez, luego le volvemos a colocar la función *new*.



Notarás que la clase *Car* tiene una propiedad "*owner*" de tipo *Contact*, pero *Contact*, en cambio, tiene una "*cars*" de tipo *NSSet*. Pero nosotros sabemos que el Set "*cars*" solo contendrá objetos de tipo *Car*, por lo que simplemente tendremos que hacer un cast cada vez que interactuemos con ellos.

No olvides modificar la clase a la que esta conectada la entidad en el modelo de la base de datos como tuvimos que hacer con la entidad *Contact*.



Si ahora corres la aplicación te darás cuenta que se cae, esto pasa porque modificamos el modelo de la base de datos. Al abrir la aplicación intenta acceder a la base de datos con el nuevo modelo, pero como este es distinto al modelo anterior se cae. Hay maneras de hacer que la base de datos se actualice y migre automáticamente a una nueva, lo cual veremos más adelante. Por ahora, borra la aplicación de tu teléfono o simulador y vuelve a correrla y debería funcionar bien.

Si nos fijamos en la clase *Contact*, la propiedad autos es un *NSSet*, es decir, es un set inmutable. Para poder modificarlo tenemos que cambiar de un *NSSet* a un *NSMutableSet*.

Tenemos dos opciones: Crear una función en la clase *Contact* que nos permita darle o quitarle un auto o modificar directamente la propiedad para que sea *NSMutableSet* para poder agregar y quitar un *Car* directamente. En este tutorial haremos la primera.

Vamos a la clase *Contact* y creamos una nueva función llamada *giveCar* y una llamada *takeCar* que nos permitan agregar un nuevo *Car* y quitar un *Car* existente:

```
func giveCar(car:Car) {
    var set = cars as! Set<Car>
    set.insert(car)
    car.assignOwner(self)
    cars = set
}
```

```
func takeCar(car:Car) {
    var set = cars as! Set<Car>
    if set.contains(car) {
        set.remove(car)
        car.deleteOwner()
    }
    cars = set
}
```

Además a la clase *Car* le damos las funciones *assignOwner* y *deleteOwner*. Un *Car* puede no tener dueño por lo que ponemos el atributo *owner* con el tipo *Contact?* como opcional.

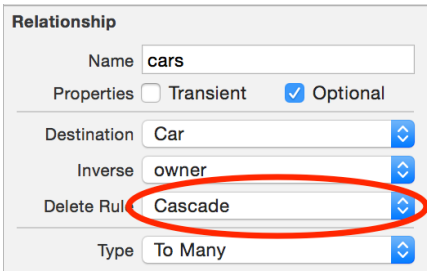
```
func deleteOwner() {
    owner = nil
}

func assignOwner(contact:Contact) {
    owner = contact
}
```

Lo único que no está faltando es crear la función *new* de la clase *Car* que llamaremos cada vez que creemos un nuevo *Car:*

```
class func new(moc: NSManagedObjectContext, manufacturer:String,
       model:String, year:String, mileage:Int) -> Car {
    let newCar = NSEntityDescription.insertNewObjectForEntityForName("Car",
        inManagedObjectContext: moc) as! EjemploCoreData.Car

    newCar.manufacturer = manufacturer
    newCar.model = model
    newCar.year = year
    newCar.mileage = mileage

    return newCar
}
```

Con esto ya conoces lo básico de relaciones y ya puedes hacer una base de datos bastante compleja. Para probarlo puedes crear una nueva vista que sea como la del ejemplo pero que cumpla con el propósito de crear *cars*.

Te podrás fijar en el ejemplo que en el modelo de la base de datos, en la entidad *Contact*, la relación *cars* tiene una Delete Rule de tipo Cascade. Esto significa que si borras un *Contact*, todos sus autos también se borraran.



Como modificar la base de datos: Migración

Cuando modificamos nuestra base de datos CoreData ya no puede accederla, ¿Por qué? Core Data busca lograr la persistencia de las entidades a través de un modelo de datos, si este modelo de datos cambia Core Data no puede accederlo (los modelos son distintos), por lo tanto no puede cumplir su función y se cae.

Para evitar este problema se usan las migraciones. Una migración consiste, en palabras simples, pasar toda la información que tenemos de un modelo de datos a otro. Dependiendo del tamaño y complejidad de esto, la migración puede ser de tipo lightweight migration o custom migration. Además, si la cantidad de información a migrar es muy grande se pueden hacer una migración por "trozos". Para este tutorial nos enfocaremos en *lightweight migration*.

Lo primero que tenemos que hacer es decirle a Xcode que queremos que haga migraciones automáticamente (fundamental para este tipo), ello modificando la variable *persistentStoreCoordinator* de nuestro *AppDelegate.swift*.

Lo que tenemos que hacer es crear la opción de migración automática:

```
let mOptions = [NSMigratePersistentStoresAutomaticallyOption: true,
NSInferMappingModelAutomaticallyOption: true]
```

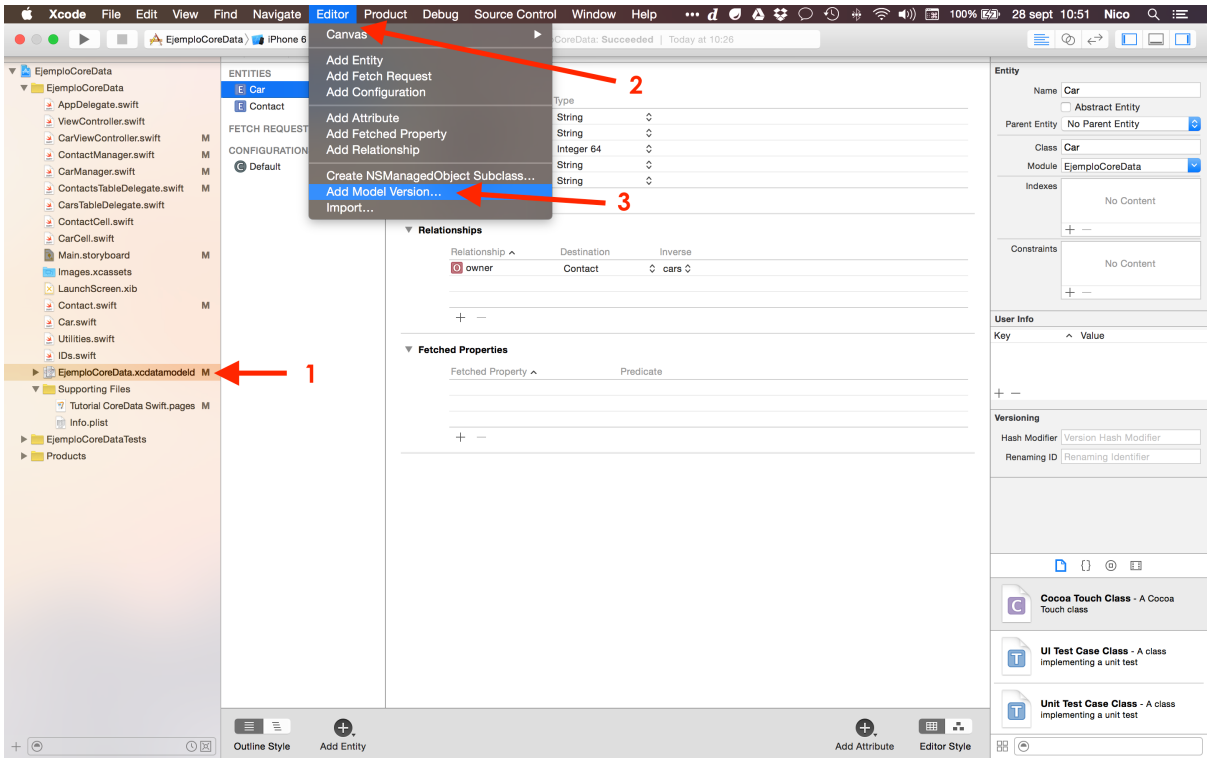
Y tomar en cuenta esta opción en la carga de nuestro *persistentStoreCoordinator*:

```
lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator? = {
    var coordinator: NSPersistentStoreCoordinator? =
NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url =
self.applicationDocumentsDirectory.URLByAppendingPathComponent("EjemploCoreData.sql
ite")
    var error: NSError? = nil
    var failureReason = "There was an error creating or loading the application's
saved data."
    let mOptions = [NSMigratePersistentStoresAutomaticallyOption: true,
NSInferMappingModelAutomaticallyOption: true]
    if coordinator!.addPersistentStoreWithType(NSSQLiteStoreType, configuration:
nil, URL: url, options: mOptions, error: &error) == nil {
        coordinator = nil
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's
saved data"
        dict[NSLocalizedFailureReasonErrorKey] = failureReason
        dict[NSUnderlyingErrorKey] = error
        error = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        NSLog("Unresolved error \(error), \(error!.userInfo)")
        abort()
    }

    return coordinator
}()
```

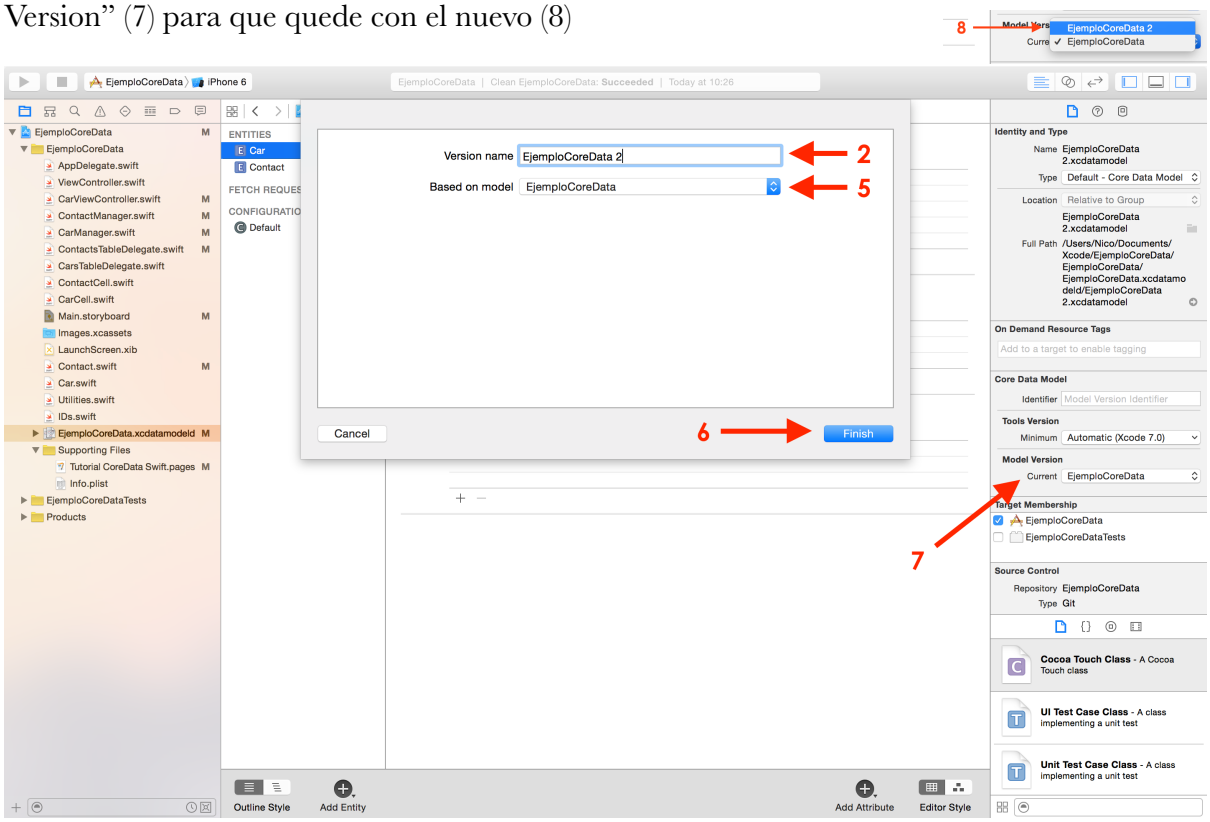
Lo que hace *mOptions* es decirle a Xcode que al encontrarse con modelos distintos debe intentar hacer la migración automáticamente.

Veamos esto en acción.

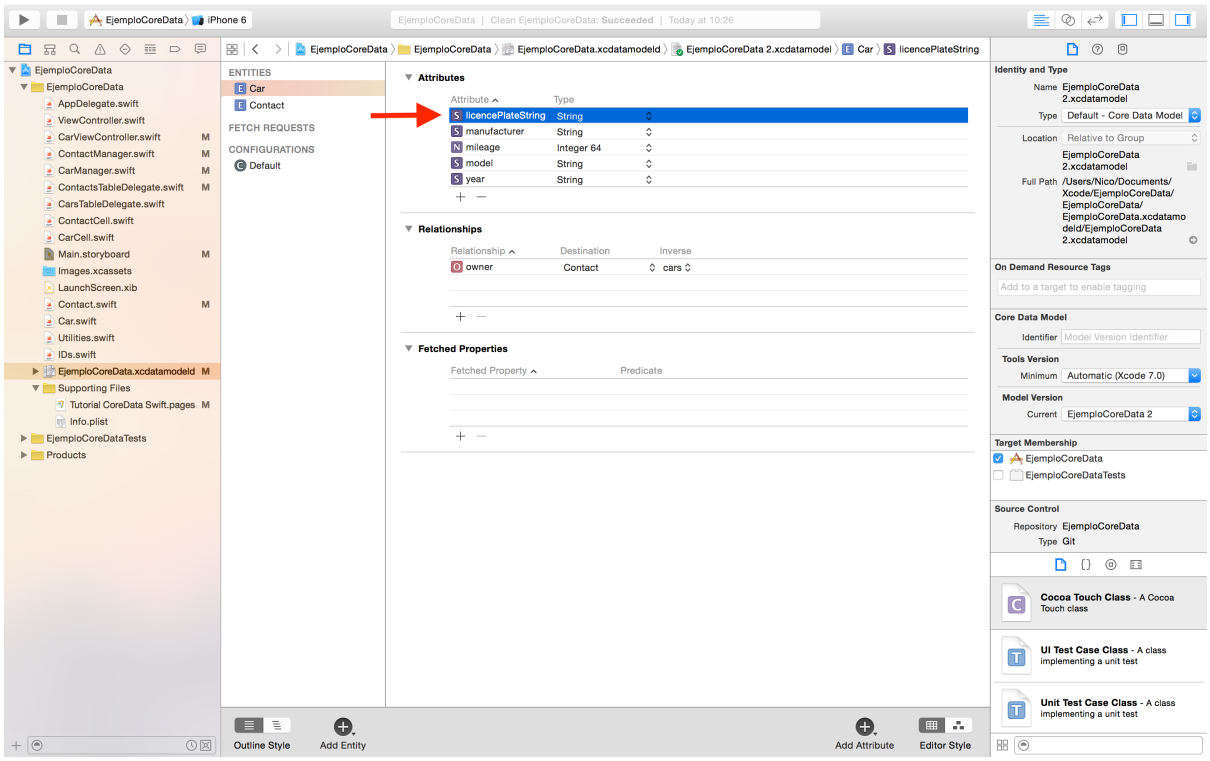
Primero, vamos a nuestro modelo actual de datos (1), lo seleccionamos, abrimos el menu del editor (2) y hacemos clic en "Add Model Version" (3)



Luego creamos el nuevo modelo (4) basado en nuestro primer modelo (5) y confirmamos (6). Después cambiamos nuestro modelo actual al nuevo modificando "Model Version" (7) para que quede con el nuevo (8)



Ahora modifiquemos nuestro nuevo modelo agregando un nuevo atributo a la entidad Auto. Lo llamaremos *licencePlateString* y será de tipo String



No olvidemos agregar esta nueva propiedad a nuestra clase *Car*

```
@NSManaged var licenPlateString: String
```

Junto con una nueva función para acceder a esta y poder modificarla, esto bajo la lógica de que un *Car* no tiene necesariamente una *licencePlate*, por lo que no es necesario que esté definida al momento de crear el *Car*.

```
var licencePlate: String {
    get { return licencePlateString }
    set { licencePlateString = newValue }
}
```

Si ahora corremos la aplicación veremos que no se cae, Xcode automáticamente migró la información del modelo de datos nº1 al modelo de datos nº2.

Una mejora que podemos hacer ahora es que al eliminar un *Contact* o *Car* se actualice la tabla respectiva automáticamente.

Para ello vamos a nuestro *ContactsTableDelegate* y agregamos una referencia al table view donde se muéstrala información proporcionada por este delegate:

```
weak var refContactsTable: UITableView!
```

Y hacemos lo mismo en *CarsTableDelegate:*

```
weak var refCarsTable : UITableView!
```

Luego, modificamos la función *commitEditingStyle* de nuestro *ContactsTableDelegate* para que actualice automáticamente la lista de contactos y muestre actualizada:

```
func tableView(tableView: UITableView,
       commitEditingStyle editingStyle: UITableViewCellEditingStyle,
       forRowAtIndexPath indexPath: NSIndexPath) {
    if (editingStyle == .Delete) {
        refContactManager.deleteContact(indexPath.row)
        refContactManager.fetchContacts()
        refContactsTable.deleteRowsAtIndexPaths([indexPath],
      withRowAnimation: .Automatic)
    }
    tableView.setEditing(false, animated: true)
}
```

Y hacemos lo mismo en *CarsTableDelegate:*

```
func tableView(tableView: UITableView,
       commitEditingStyle editingStyle: UITableViewCellEditingStyle,
       forRowAtIndexPath indexPath: NSIndexPath) {
    if (editingStyle == .Delete) {
        refCarManager.deleteCar(indexPath.row)
        refCarManager.fetchCars()
        refCarsTable.deleteRowsAtIndexPaths([indexPath],
             withRowAnimation: .Automatic)
    }
    tableView.setEditing(false, animated: true)
}
```

Y en las clases *ViewController* le damos al delegate la referencia a la tabla:

```
contactsTableDelegate.refContactsTable = ContactsTable
```

Y en las clases *CarViewController* también:

```
carsTableDelegate.refCarsTable = CarsTable
```

Asegurándonos de colocarlas después de haber instanciado el delegate.

De esta manera la tabla se actualizará automáticamente cuando se borre algún objeto.

También, si quieres, puedes mejorar la aplicación asegurando que en el atributo kilometraje se acepten solo números y que se muestre una alerta si esto no ocurre.

Puedes incluso modificar el modelo para que el año de un auto y el número de un contacto sean Int y revisar cuando uno ingresa estos datos para que estén correctos. El código de ejemplo solo revisa el ingreso del kilometraje.

Puedes descargar el código fuente final mas actualizado aquí. Si algo no funciona como debería, compara tu proyecto con el descargado. ¡Ya estás preparado para utilizar Core Data!

Glosario

Core Data: Framework de object graph y persistencia de Apple para base de datos.

Managed Object: Son instancias de una entidad de nuestra base de datos.

Managed Object Context: Es el contexto en el cual se encuentran los Managed Objects. Es una representación de lo que esta contenido en el Store.

NSManaged: Atributo que indica que los métodos de inicialización de una propiedad serán proveídos por el sistema en tiempo de ejecución.

Persistence Store Coordinator: Se encarga de coordinar el Persistence Object Store para poder mostrarlo en un Managed Object Context.

Persistence Object Store: Se encarga de mantener un mapa entre los objetos de una aplicación con los records en un Persistent store.

Persistence Store: Un archivo de base de datos donde cada record tiene las últimas instancias guardadas de un Managed Object. Tiene 3 tipos nativos: binary, XML y SQLite