

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

Tabla de contenidos	1
¿Por qué usar una base de datos? ¿Qué es?	2
Creando la base de datos	2
Una descripción breve de Core Data	3
Como guardar información	3
Definiendo las entidades	4
Preparándose para interactuar con la base de datos	7
Agregar un nuevo contacto a la base de datos	8
Obtener los contactos de la base de datos	9
Como borrar objetos de la base de datos	11
Búsqueda avanzada de objetos (fetch con predicate)	11
Ordenando los resultados del fetch	12
Filtrando los resultados del fetch	12
Entidades cuyos atributos son otras entidades: Relaciones	14
Como modificar la base de datos: Migración	17
Glosario	22

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

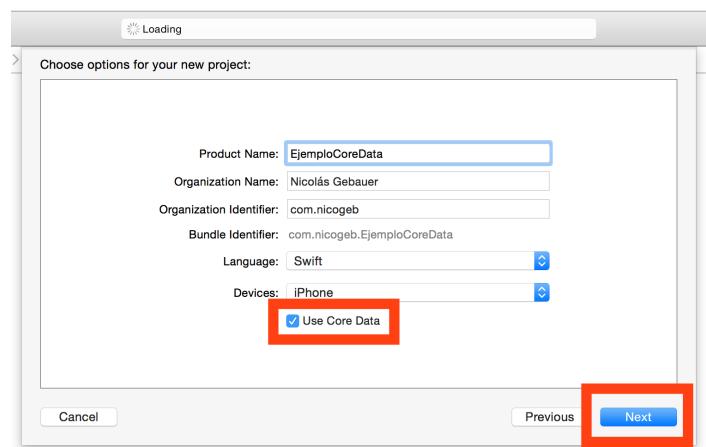
Una base de datos se usa cuando uno quiere guardar información que se mantenga a través de varias instancias de la aplicación, incluso cuando se cierre o se apague el dispositivo, a esto se le llama persistencia. Uno podría guardar todo en un archivo de texto con algún formato, como por ejemplo, “Nicolás;Gebauer;Martínez” 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.

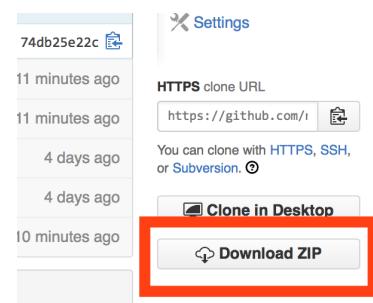
Empecemos con la parte importante ahora, ¿Cómo se usa?

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 está 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”.



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.

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 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.

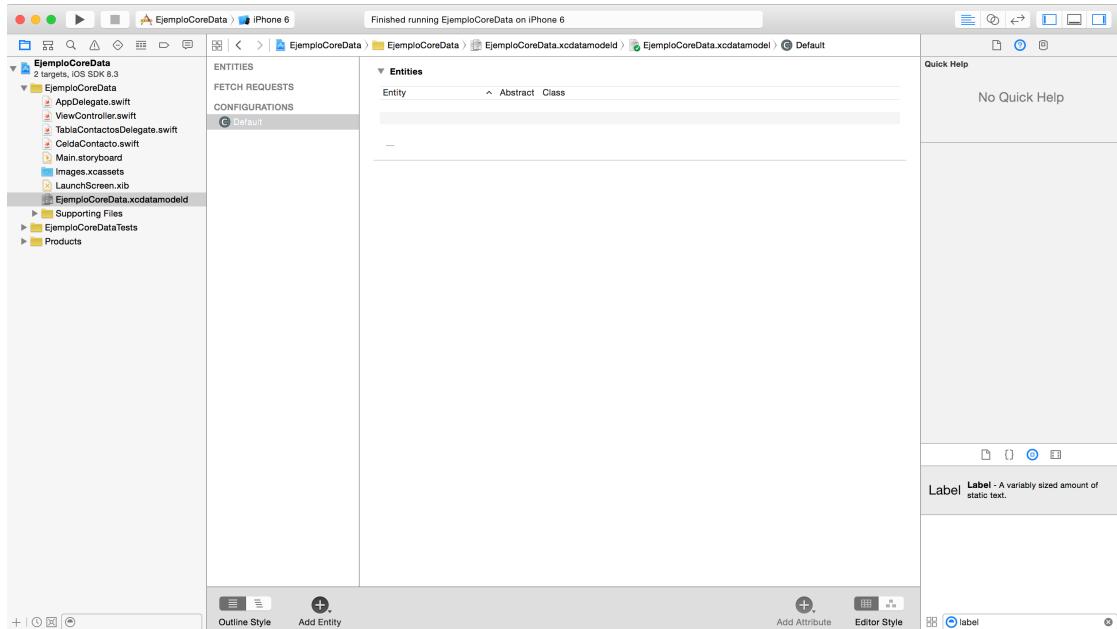
Puedes leer más sobre CoreData [aquí](#).

Como guardar información

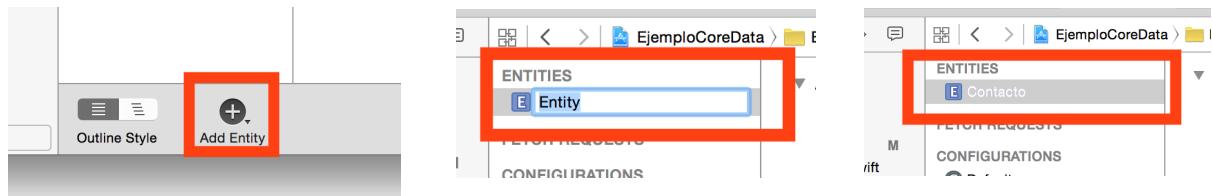
En CoreData lo que se guardan son entidades, instancias de un objeto. Estas se guardan con sus atributos y pueden ser obtenidas de la base de datos después. Antes de poder crear/guardar/obtener entidades, debemos decirle a la base de datos que entidades guardará, que atributos tendrán y que relaciones habrá entre ellas.

Definiendo las entidades

Para definir las entidades que usará la base de datos usamos el modelo “EjemploCoreData.xcdatamodeld”. Al abrirlo deberías ver algo así:

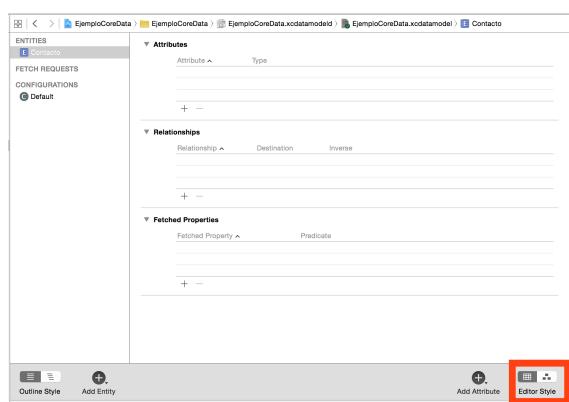


Para este tutorial haremos una base de datos que guarde contactos. Primero, agregamos una nueva entidad haciendo clic en “Add Entity” y cambiamos su nombre a “Contacto”.

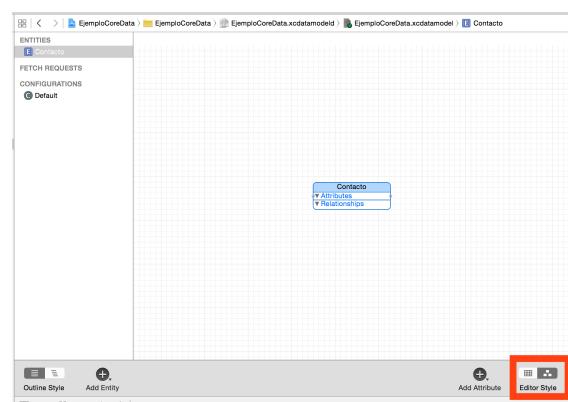


Nota: Existen dos estilos del editor, uno tipo tabla y uno gráfico, en este tutorial usaremos el editor de tipo tabla.

Editor tipo tabla



Editor gráfico



Nuestro Contacto tendrá 3 atributos: Un nombre, un apellido y un número. Para ello seleccionamos la entidad Contacto y hacemos clic en “Add attribute” (1) 3 veces, luego modificamos el nombre de cada atributo (2) y le asignamos el tipo “String” (3).

The screenshot shows two instances of the Xcode Core Data editor interface, illustrating the process of adding and modifying attributes for the 'Contacto' entity.

Top Screenshot (Initial State):

- Entity List:** Shows 'ENTITIES' and 'Contacto' selected.
- Attributes Section:**
 - Header: Attribute ▾ Type
 - Row 1: **attribute** (highlighted with red box 2) **Undefined** (highlighted with red box 3)
 - Row 2: attribute1 Undefined
 - Row 3: attribute2 Undefined
 - Action Bar: A red circle labeled '1' surrounds the '+' button.
- Relationships Section:** Empty.
- Fetched Properties Section:** Empty.
- Toolbar:** Includes 'Outline Style', 'Add Entity', 'Add Attribute' (highlighted with a red box), and 'Editor Style'.

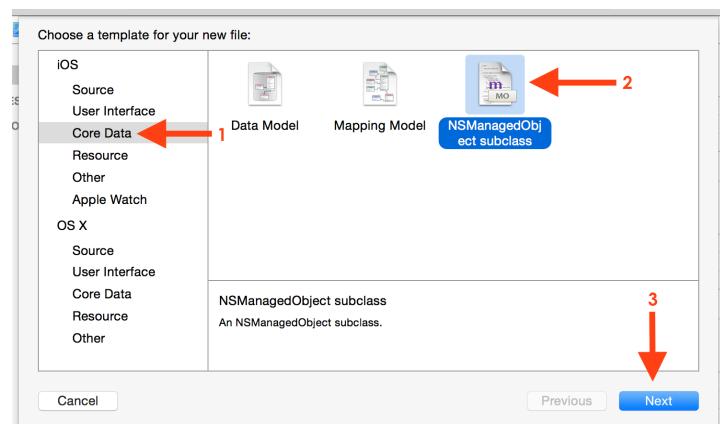
Bottom Screenshot (Final State):

- Entity List:** Shows 'ENTITIES' and 'Contacto' selected.
- Attributes Section:**
 - Header: Attribute ▾ Type
 - Row 1: **nombre** String (highlighted with red box 1)
 - Row 2: apellido String
 - Row 3: numero String
 - Action Bar: A red circle labeled '1' surrounds the '+' button.
- Relationships Section:** Empty.
- Fetched Properties Section:** Empty.
- Toolbar:** Includes 'Outline Style', 'Add Entity', 'Add Attribute' (highlighted with a red box), and 'Editor Style'.

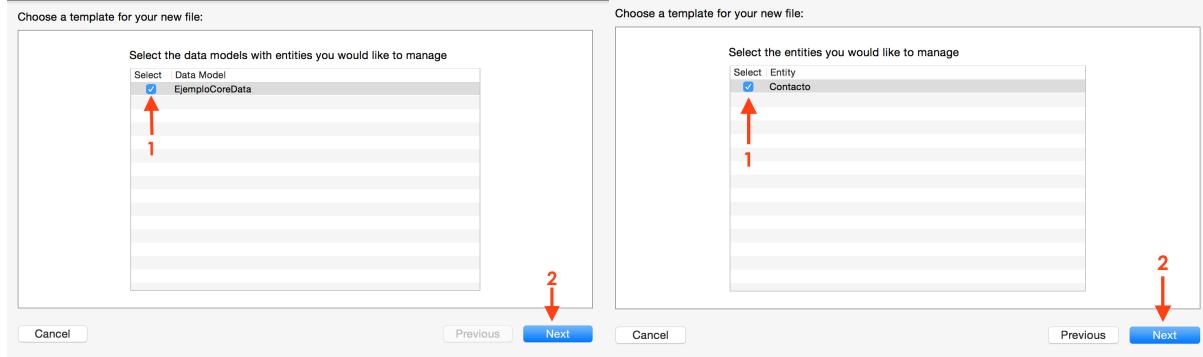
In both screenshots, the 'Add Attribute' button in the toolbar is highlighted with a red box, indicating the step where new attributes are added. The first attribute is named 'attribute' and is later modified to 'nombre'. The type is changed from 'Undefined' to 'String'.

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.

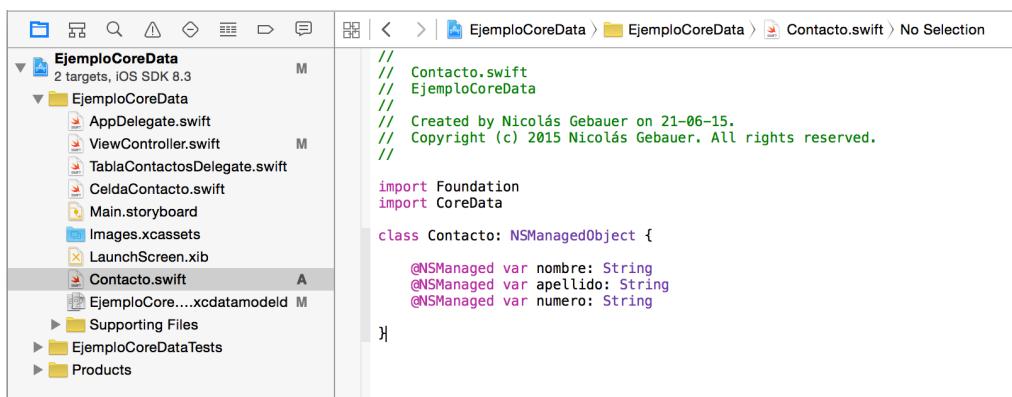
Para poder trabajar con las entidades guardadas en la base de datos o crear nuevas necesitamos una clase para cada entidad. Para crear esta clase de una manera rápida y automática creamos un nuevo archivo (Cmd + N), seleccionamos la opción “Core Data” (1), “NSManagedObject subclass” (2) y siguiente (3).



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

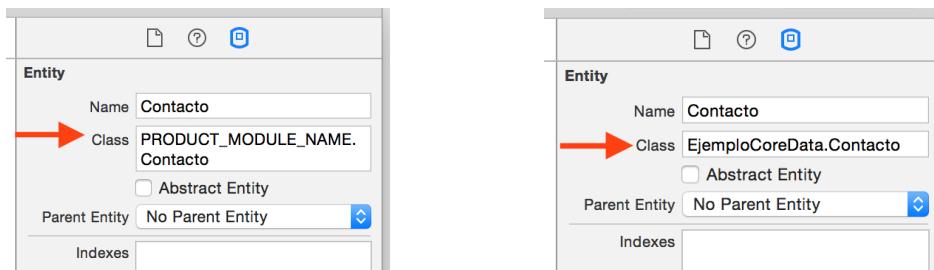


Y la clase Contacto se habrá creado mágicamente.



Los atributos de esta clase tienen un nuevo sufijo @NSManaged, para saber que hace esta propiedad prueba a borrarla. Verás que da un error de tipo “Contacto no tiene un inicializador” o “La propiedad necesita un valor inicial”. @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 interactúas 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 Contacto 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 “Aregar contacto” y “Fetch contactos”, por lo que ella tendrá una instancia de ContactManager referenciada. También, le dará una referencia de esta a “TablaContactosDelegate” (el delegate) para poder cargar los datos de los contactos.

Agregamos el siguiente atributo a ViewController:

```
let managerContactos = ContactManager()
```

Y reemplazamos la función viewDidLoad por esto:

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

    delegateTablaContactos = TablaContactosDelegate()
    delegateTablaContactos.contactManager = managerContactos
    TablaContactos.delegate = delegateTablaContactos
    TablaContactos.dataSource = delegateTablaContactos
}
```

Y al delegate le agregamos el siguiente atributo

```
weak var contactManager : ContactManager!
```

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

Aregar 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 ella. Esta función recibirá un nombre, un apellido y un número como argumentos. Para ello vamos a la clase Contacto y agregamos la siguiente función

```
class func new(moc: NSManagedObjectContext, _nombre:String, _apellido:String,
    _numero:String) -> Contacto {
    let newItem = NSEntityDescription.insertNewObjectForEntityForName("Contacto",
        inManagedObjectContext: moc) as! EjemploCoreData.Contacto
    newItem.nombre = _nombre
    newItem.apellido = _apellido
    newItem.numero = _numero

    return newItem
}
```

Esta función creará una nueva instancia de Contacto, la guardará en la base de datos y luego la retornara para poder usarla y/o modificarla.

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

En el ContactManager creamos una nueva función:

```
func agregarNuevoContacto(nombre:String, apellido:String, numero:String) {  
    let contacto = Contacto.new(moc!, _nombre: nombre, _apellido: apellido,  
    _numero: numero)  
}
```

Y la llamamos desde el ViewController en la acción AgregarNuevoContacto:

```
@IBAction func AgregarNuevoContacto(sender: AnyObject) {  
    managerContactos.agregarNuevoContacto(LabelNombre.text, apellido:  
LabelApellido.text, numero: LabelNumero.text)  
    borrarLabels()  
}
```

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

```
func saveDatabase() {  
    var error : NSError?  
    if(moc!.save(&error) ) {  
        if let err = error?.localizedDescription {  
            NSLog("Error grabando: ")  
            NSLog(error!.localizedDescription as String)  
        }  
    }  
}
```

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.

Y agregamos una nueva línea en la función agregarNuevoContacto para grabar la base de datos, con lo cual nos quedaría así:

```
func agregarNuevoContacto(nombre:String, apellido:String, numero:String) {  
    let contacto = Contacto.new(moc!, _nombre: nombre, _apellido: apellido,  
    _numero: numero)  
    saveDatabase()  
}
```

¡Felicidades! Puedes agregar nuevos contactos a tu base de datos y estos quedarán guardados aunque cierres la aplicación y/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 a nuestra clase ContactManager:

```
var contactos = [Contacto]()
```

Y creamos una nueva función llamada fetchContacts de la siguiente forma:

```
func fetchContacts() {
    let fetchRequest = NSFetchRequest(entityName: "Contacto")
    if let fetchResults = moc!.executeFetchRequest(fetchRequest, error: nil) as?
    [Contacto] {
        contactos = fetchResults
    }
}
```

Ahora cada vez que hagamos fetchContacts nuestro array 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, para ello modificaremos las siguientes funciones en nuestro delegate:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let c = tableView.dequeueReusableCellWithIdentifier("IDCeldaContacto") as!
    CeldaContacto
    let contacto = contactManager.contactos[indexPath.row]
    c.LabelNumero.text = contacto.numero
    c.LabelNombre.text = contacto.nombre + " " + (contacto.apellido)
    return c
}

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

Ahora solo falta que al hacer clic en Fetch contactos estos se actualicen, para ello vamos a la función FetchContactos en nuestro ViewController y agregamos lo siguiente:

```
@IBAction func FetchContactos(sender: AnyObject) {
    managerContactos.fetchContacts()
    TablaContactos.reloadData()
}
```

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

Si tu aplicación no funciona como debería, o no estás seguro de algo que escribiste, puedes descargar el código fuente [aquí](#) y compararlo con lo que tienes en tu proyecto.

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 index. La función se vería así:

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

Para llamarla desde nuestro delegate debemos modificar dos funciones:

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

func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
    if(editingStyle == .Delete ) {
        contactManager.borrarContacto(indexPath.row)
    }
    tableView.setEditing(false, animated: true)
}
```

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

Si tu aplicación no funciona como debería, o no estás seguro de algo que escribiste, puedes descargar el código fuente [aquí](#) y compararlo con lo que tienes en tu proyecto.

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 cumplen 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 quedar así:

```
func fetchContactos() {
    let sortDescriptor = NSSortDescriptor(key: "nombre", ascending: true)
    let fetchRequest = NSFetchedResultsController(entityName: "Contacto")
    fetchRequest.sortDescriptors = [sortDescriptor]
    if let fetchResults = moc!.executeFetchRequest(fetchRequest, error: nil) as?
        [Contacto] {
        contactos = fetchResults
    }
}
```

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

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” haríamos un llamado así:

```
let predicate = NSPredicate(format: "nombre CONTAINS %@", "co")
let fetchRequest = NSFetchedResultsController(entityName: "Contacto")
fetchRequest.predicate = predicate
if let fetchResults = moc!.executeFetchRequest(fetchRequest, error: nil) as?
    [Contacto] {
    contactos = fetchResults
}
```

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

Además, si queremos usar varios predicates haríamos un llamado así:

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

En este caso contactos sería un array con los contactos cuyo nombre contiene “co” y cuyo apellido 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 nombre contiene “co” o cuyo apellido 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 y modificaremos la función FetchContactos del ViewController para reaccionar cuando haya filtros que aplicar.

La nueva función de nuestro ContactManager sería así:

```
func fetchContactsWithPredicates(nombre:String, apellido:String, numero:String) {
    var predicatesArray = [NSPredicate]()
    if nombre != "" {
        let predicateNombre = NSPredicate(format: "nombre CONTAINS %@", nombre)
        predicatesArray.append(predicateNombre)
    }
    if apellido != "" {
        let predicateApellido = NSPredicate(format: "apellido CONTAINS %@", apellido)
        predicatesArray.append(predicateApellido)
    }
    if numero != "" {
        let predicateNumero = NSPredicate(format: "numero CONTAINS %@", numero)
        predicatesArray.append(predicateNumero)
    }
    let predicate = NSCompoundPredicate(type:
NSCompoundPredicateType.AndPredicateType, subpredicates: predicatesArray)
    let sortDescriptor = NSSortDescriptor(key: "nombre", ascending: true)
    let fetchRequest = NSFetchedResultsController(entityName: "Contacto")
    fetchRequest.sortDescriptors = [sortDescriptor]
    fetchRequest.predicate = predicate
    if let fetchResults = moc!.executeFetchRequest(fetchRequest, error: nil) as?
[Contacto] {
        contactos = fetchResults
    }
}
```

La función FetchContactos quedaría así:

```
@IBAction func FetchContactos(sender: AnyObject) {
    if (LabelNombre.text != "" || LabelApellido.text != "" || LabelNumero.text != "") {
        managerContactos.fetchContactsWithPredicates(LabelNombre.text, apellido:
LabelApellido.text, numero: LabelNumero.text)
    }
    else {
        managerContactos.fetchContacts()
    }
    borrarLabels()
    TablaContactos.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 contactos.

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.

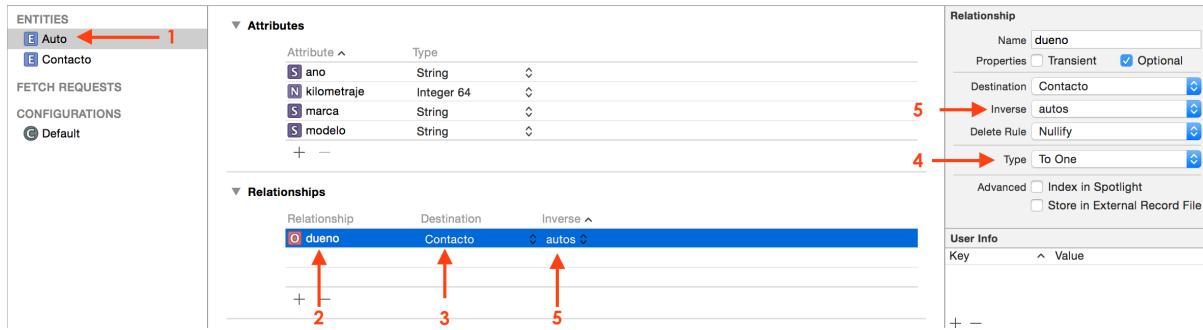
Las relaciones se refieren a entidades que se relacionan con otras entidades, ya sea que una entidad contiene muchas entidades o que dos entidades están estrechamente relacionadas. Imaginemos que cada Contacto tiene un auto y cada auto tiene sus propias características como marca, año, kilometraje, etc.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

Attribute	Type
S año	String
N kilometraje	Integer 64
S marca	String
S modelo	String

Ahora, cada contacto podría tener uno o más autos, así que vamos a la entidad Contacto (1), agregamos una nueva relación que se llamará “autos” (2), su destino será la entidad auto (3) y su relación será de tipo To Many (3).

Luego debemos hacer la relación inversa, es decir; del auto al Contacto. Cada auto tendrá un solo dueño, por lo que vamos a la entidad auto (1), agregamos una nueva relación

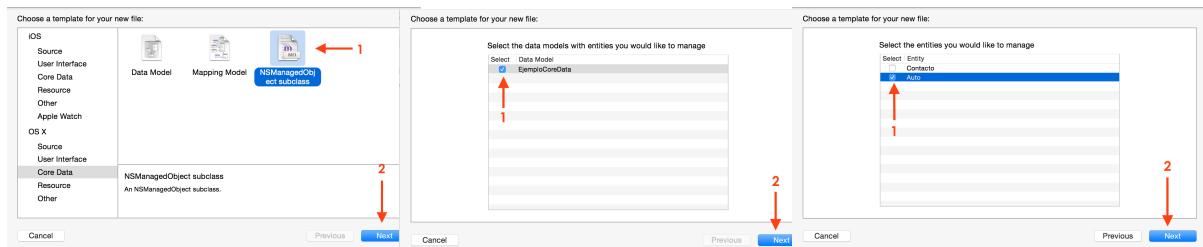
que se llamará “dueno” (2), su destino será la entidad contacto (3) y su relación será de tipo



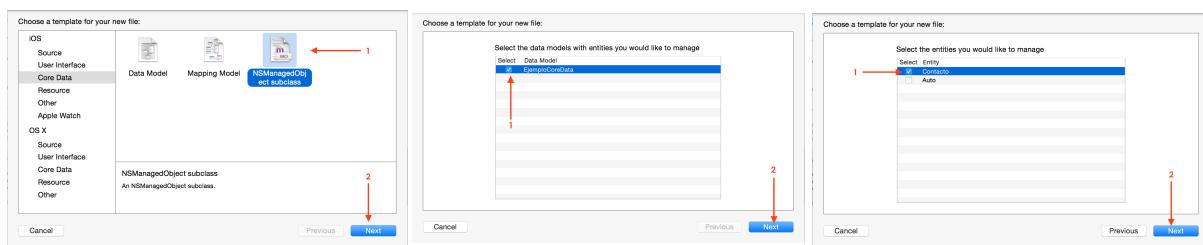
To One (4), además, su inversa será la relación “autos” (5).

Ahora cuando volvemos a la entidad contacto te darás cuenta que la relación inversa se agregó automáticamente. Esto significa que cada contacto puede tener uno o más autos, cada uno con sus propias propiedades, pero cada auto solo puede tener un dueño (contacto).

Ahora creamos la nueva clase Auto y actualicemos nuestra clase Contacto. Creamos una nueva clase de tipo NSManagedObject subclass para la entidad Auto.

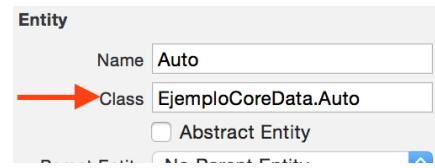


Ahora actualizamos la clase Contacto, para ello guardamos la función new y volvemos a crear la clase al igual que la primera vez, luego le volvemos a colocar la función new.



Notarás que la clase Auto tiene una propiedad “dueno” de tipo Contacto, pero Contacto, en cambio, tiene una “autos” de tipo NSSet. Pero nosotros sabemos que el Set “autos” solo contendrá objetos de tipo Auto, 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 Contacto.



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/simulador y vuelve a correrla y debería funcionar bien.

Si nos fijamos en la clase Contacto, la propiedad autos es un NSSet, es decir, es un set fijo. Tenemos dos opciones: Crear una función en la clase Contacto que nos permita darle/quitarle un auto o modificar la propiedad para que sea NSMutableSet para poder agregar/quitar autos directamente. En este tutorial haremos la primera.

Vamos a la clase Contacto y creamos una nueva función llamada darAuto y una llamada quitarAuto de la siguiente manera:

```
func darAuto(auto:Auto) {
    var set = autos as! Set<Auto>
    set.insert(auto)
    auto.asignarDueño(self)
    autos = set
}

func quitarAuto(auto:Auto) {
    var set = autos as! Set<Auto>
    if set.contains(auto) {
        set.remove(auto)
        auto.eliminarDueño()
    }
    autos = set
}
```

Además a la clase Auto le damos las funciones asignarDueño y eliminarDueño. También modificamos la propiedad dueño para que sea de tipo Contacto? ya que puede darse el caso de que un auto no tenga dueño.

```
func eliminarDueño() {
    dueño = nil
}

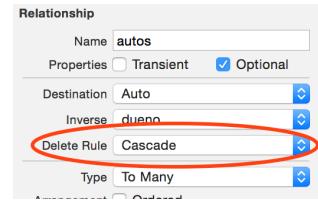
func asignarDueño(contacto:Contacto) {
    dueño = contacto
}
```

Lo único que no está faltando es crear la función new de la clase Auto que llamaremos cada vez que creemos un nuevo auto. Se debería ver así:

```
class func new(moc: NSManagedObjectContext, _marca:String, _modelo:String,  
_ano:String, _kilometraje:Int = 0) -> Auto {  
    let newAuto = NSEntityDescription.insertNewObjectForEntityForName("Contacto",  
inManagedObjectContext: moc) as! EjemploCoreData.Auto  
  
    newAuto.marca = _marca  
    newAuto.modelo = _modelo  
    newAuto.ano = _ano  
    newAuto.kilometraje = _kilometraje  
  
    return newAuto  
}
```

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 autos. Puedes ver un ejemplo funcionando [aquí](#).

Te podrás fijar en el ejemplo que en el modelo de la base de datos, en la entidad Contacto, la relación autos tiene una Delete Rule de tipo Cascade. Esto significa que si borras un Contacto, 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), para ello vamos a nuestro AppDelegate.swift y hacemos unos cambios en la variable persistentStoreCoordinator para que pase de verse así:

```
lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator? = {  
    var coordinator: NSPersistentStoreCoordinator? =  
    NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
```

```

        let url =
self.applicationDocumentsDirectory.URLByAppendingPathComponent("EjemploCoreData.sqlite")
        var error: NSError? = nil
        var failureReason = "There was an error creating or loading the application's
saved data."
        if coordinator!.addPersistentStoreWithType(NSSQLiteStoreType, configuration:
nil, URL: url, options: nil, error: &error) == nil {
            coordinator = nil
            var dict = [String: AnyObject]()
            dict[NSLocalizedStringKey] = "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
}()

```

A verse así:

```

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator? = {
    var coordinator: NSPersistentStoreCoordinator? =
NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url =
self.applicationDocumentsDirectory.URLByAppendingPathComponent("EjemploCoreData.sqlite")
    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[NSLocalizedStringKey] = "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
}()

```

Notaras que hicimos dos pequeños cambios

1. Agregamos la línea:

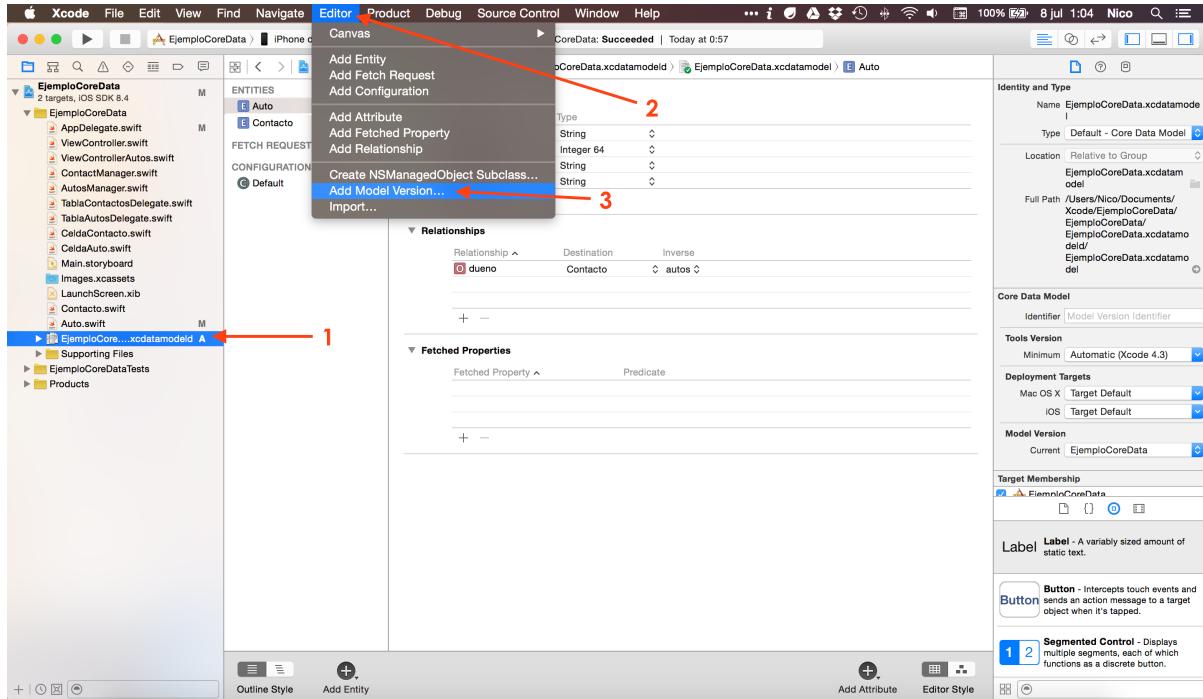
```
let mOptions = [NSMigratePersistentStoresAutomaticallyOption: true,
NSInferMappingModelAutomaticallyOption: true]
```

2. Modificamos el parámetro options de coordinator!.addPersistentStoreWithType(...)

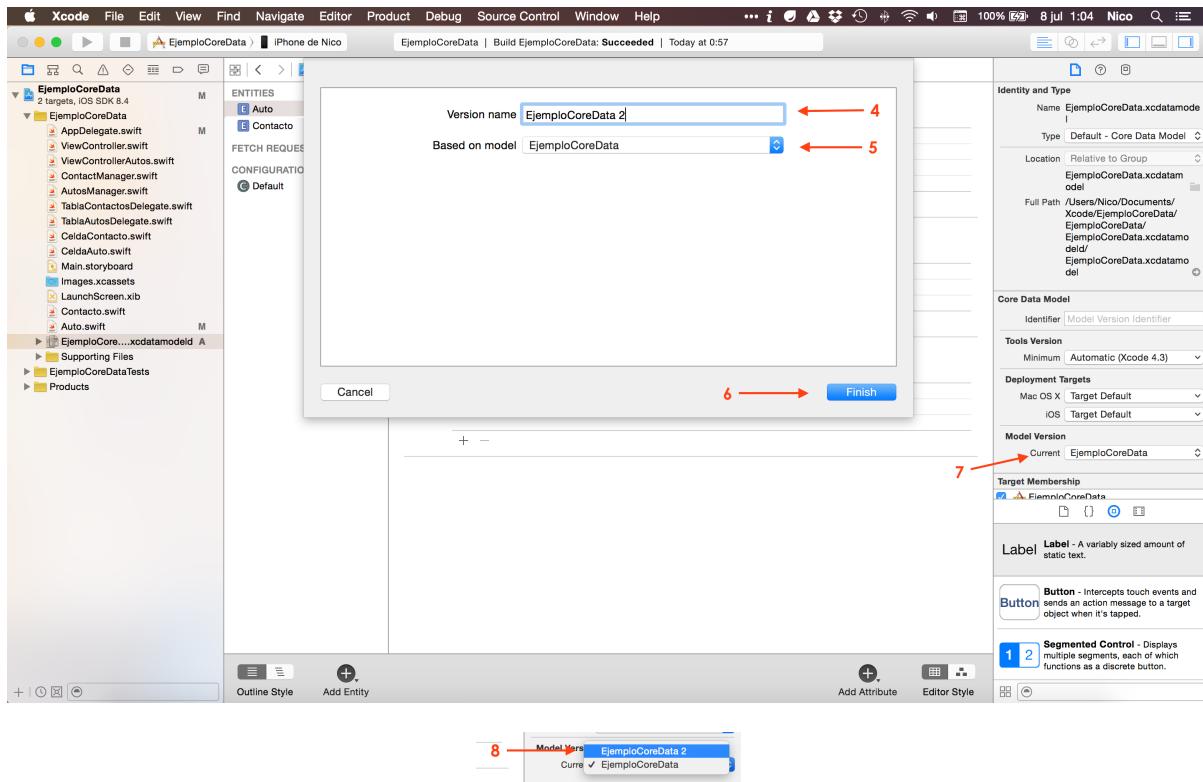
Lo que hace mOptions es decirle a Xcode que al encontrarse con modelos distintos debe intentar hacer la migración automáticamente.

Veamos esto en acción.

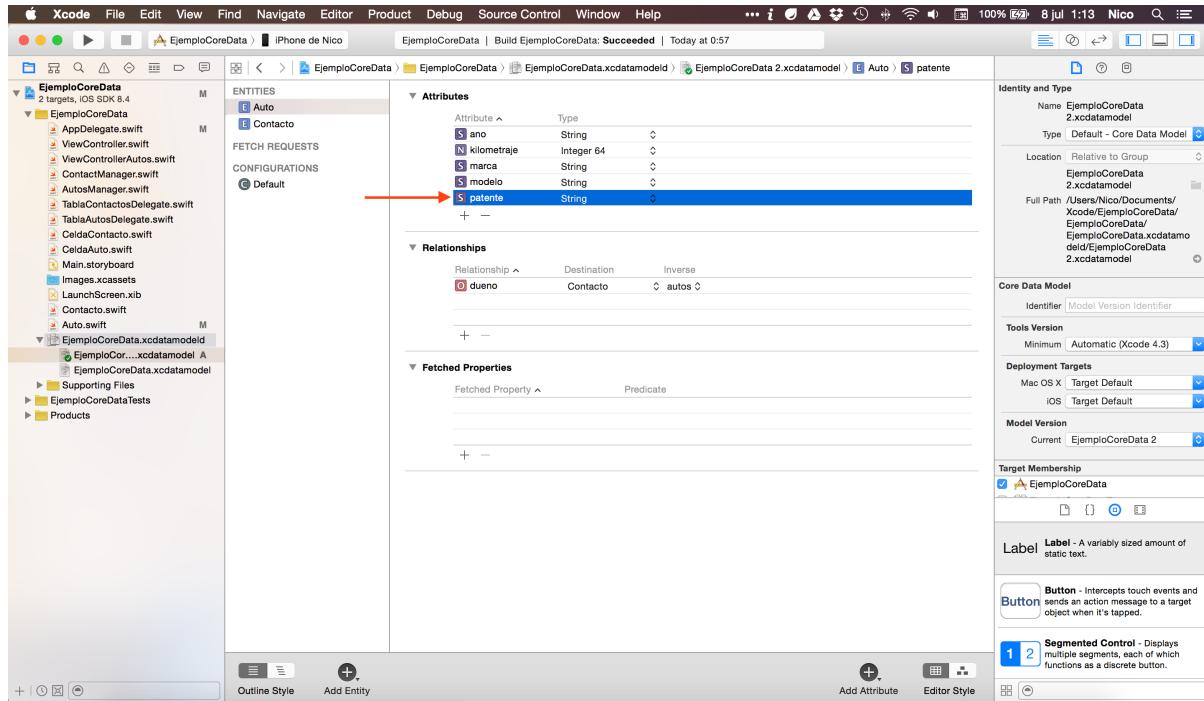
Primero, vamos a nuestro modelo actual de datos (1), lo seleccionamos, abrimos el menu del editor (2) y hacemos clic en “Add Model Version” (3)



Luego creamos el nuevo modelo (4) basado en nuestro primer modelo (5) y confirmamos (6). Despu s 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 patente y será de tipo String



No olvidemos agregar esta nueva propiedad a nuestra clase Auto

```
@NSManaged var patenteString:String
```

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

```
var patente: String {
    get { return patenteString }
    set { patenteString = 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 contacto/auto se actualice la tabla respectiva automáticamente.

Para ello vamos a nuestro TablaContactosDelegate y TablaAutosDelegate y agregamos el siguiente atributo (respectivamente):

```
weak var referenciaALaTablaContactos : UITableView!
weak var referenciaALaTablaAutos : UITableView!
```

Y modificamos la función commitEditingStyle de ambos para que queden de la siguiente forma (respectivamente):

```
func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
    if (editingStyle == .Delete) {
        contactManager.borrarContacto(indexPath.row)
        contactManager.fetchContacts()
        referenciaALaTablaContactos.reloadData()
    }
    tableView.setEditing(false, animated: true)
}

func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
    if (editingStyle == .Delete) {
        autosManager.borrarAuto(indexPath.row)
        autosManager.fetchAutos()
        referenciaALaTablaAutos.reloadData()
    }
    tableView.setEditing(false, animated: true)
}
```

Y en las clases ViewController y ViewControllerAutos agregamos una nueva línea en viewDidLoad que sea así (respectivamente):

```
delegateTablaContactos.referenciaALaTablaContactos = TablaContactos
delegateTablaAutos.referenciaALaTablaAutos = TablaAutos
```

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