**CoreData Stacks**

صورة تحتوي على نص, لقطة شاشة, كمبيوتر

تم إنشاء الوصف تلقائياً

Delete relational data in core data

<https://www.advancedswift.com/batch-delete-everything-core-data-swift/>

Notification center

<https://developer.apple.com/documentation/foundation/notificationcenter>

**Relationship**

A relationship is a link between multiple entities. In Core Data, the relationship between two entities is called   **to-one**relationships**,**while those between one and many entities are called **to-many**relationships**.**For example, a *Customer*can have a to-many relationship with a list of orders whereas an individual *Order*will have a to-one relationship with the *Customer.*

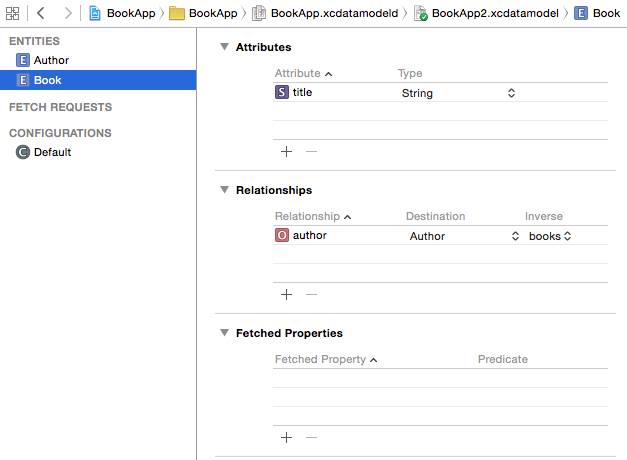
Make sure to create a new project for this demonstration and don't forget to check  *User Core Data.*Name this project as **BookApp.**

1. Open the data model file (**.datamodeld)**and create a new entity named *Author*and add an attribute *name*of type String.
2. Since we want to keep track which Author wrote each Book, define another entity and name it *Book,*then add an attribute *title*of type String.
3. Go back to the Author entity. You might think you need to add a new attribute of type Array to hold the books, but there is no Array type in Core Data. Instead, the way to do this is to model it as a *relationship.*Add a new relationship and name it *books:*

**

You can think of  *Destination*as the receiving end of a relationship.

1. Select the *Book*entity and create an inverse relationship back to *Author.*Set the destination as *author* and the inverse as *books.*

**

The inverse lets the model know how to find its way back, so to speak. Given a book record, you can follow the relationship back to the author. Thanks to the inverse, the model knows to follow the  **books**relationship to get back to the book record.

1. Every relationship is a to-one relationship by default, which means you can only track one book per author at the moment. We don't want the default relationship because we need to track more than one book to an author. To fix this, select the *books*relationship, open the Data Model tab, click on the *Type*drop-down, and select *To Many.*This means one author can write many books. Also, make sure you check the Ordered checkbox for the *books*Arrangement.

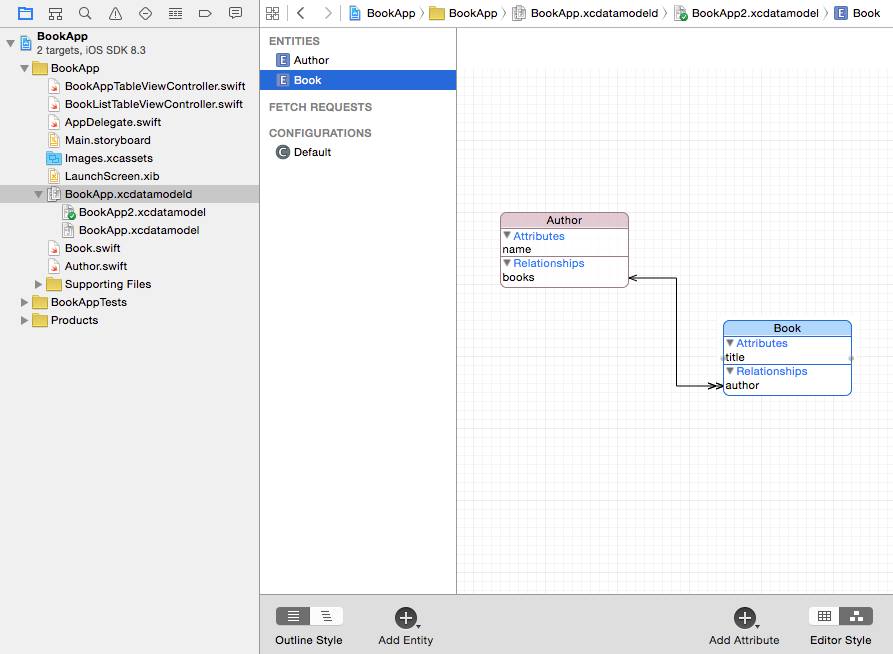
صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

NOTE: If you don't check the  *Ordered*checkbox, the default datatype of *books*will be *NSSet.*Unlike Arrays, sets don't allow accessing their members by index. In fact, there's no ordering at all! Since we want to access individual objects by index, we need to check the  *Ordered*checkbox in the visual editor, as you've done here. Core Data will then represent the relationship as an *NSOrderedSet*

**Graph Editor**

This entire time we have been looking at the table editor style for our Models. We can toggle the segmented control (located on the bottom-right) to switch to the graph editor style:



The graph editor is a great tool to visualize the relationships between your Core Data entities. Here we see the to-many relationship from Author to Book is represented with a double arrow. Book points back to Author with a single arrow (to-one relationship). Feel free to switch back and forth between two editor styles.

**Managed object subclasses**

1. Select the Book model and update the 'Module' property to 'Current Product Module.' Do the same for the Author model.
2. Save your project and run a build (Cmd + R). Try to create an instance of the Book and Author in the ViewController's viewDidLoad method: `let book = Book(), author = Author();` If you have access to the Book and Author objects, continue on to the next step. If not, close and reopen Xcode and try again.
3. Delete the line of code we just inserted into the viewDidLoad method. It was used as a simple way to check if we had access to the Book and Model objects.

Now that our Core Data is set, let's switch to our  **.storyboard:**

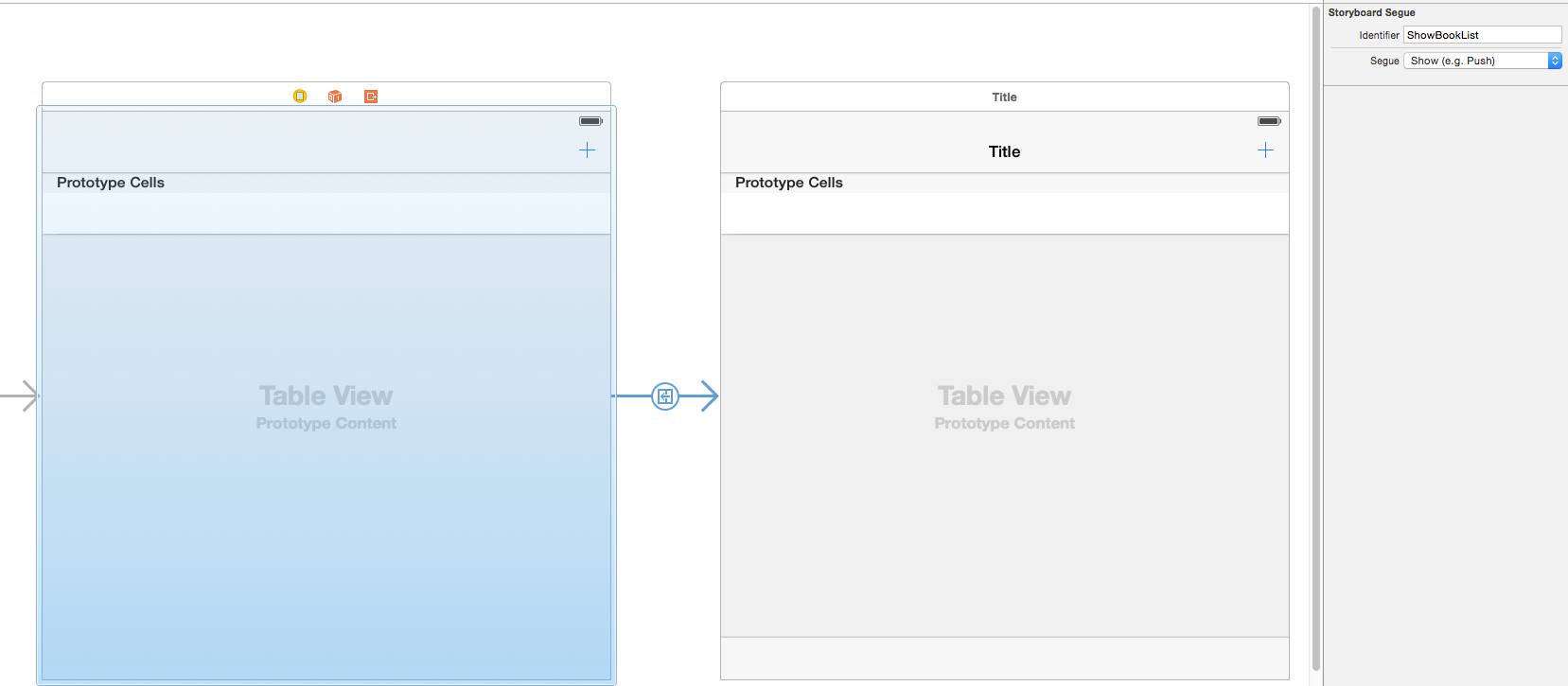
1. We will be using a Table View Controller to display a list of authors. (Refer to the **Load & Delete** tab, Journal project on how to create a basic table view controller and how to persist via form. Create at least 5 authors )
2. Create a new Cocoa Touch file and name it as **BookListTableViewController.**This will be the *show*page for the authors. The idea is, whenever users tap on an author, users will be redirected to a Book List page that shows all of the author's books. (Refer to the **Load & Delete**tab, Journal project on how to create a basic table view controller. Skip the persisting part because we will have a different approach for that.).
3. For now, we just want to see some books listed under Book List page (without persisting anything yet to Core Data). So for the*tableView(numberOfRowsInSection)*method, let's just ***return 3****,*and for the *tableView(CellForRowAtIndexPath)*method, let's just have ***cell.textLabel!.text = "Book \(indexPath.row)".***
4. Ctrl + drag from the yellow circle icon at the top of **AuthorListTableViewController**into the Book List Table View Controller and create a show segue.

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

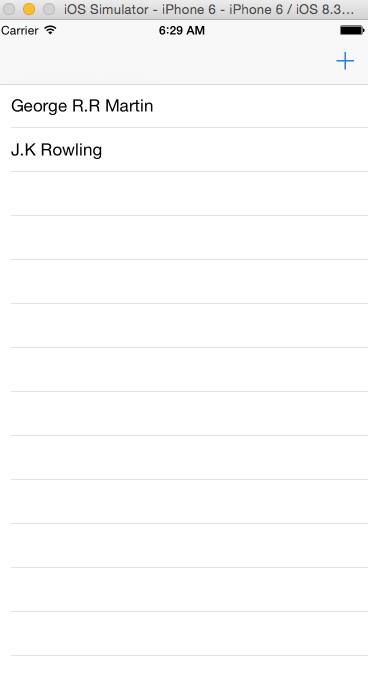
This adds a push transition from the Author List screen to the Book list screen. It also puts the navigation bar on the Book list scene. Note that this new segue isn't attached to any button or table view cell. There is nothing on the Author List screen that you can tap or otherwise interact with in order to trigger the segue. That means you have to perform it programmatically.

1. Click on the new segue to select it, go to the *Attributes inspector*and give it the identifier **ShowBookList**

****

The segue style should be Show because you're pushing the Book List View Controller onto the navigation stack when performing the segue.

1. In **AuthorListTableViewController.swift,**add the *tableView(didSelectRowAtIndexPath)*method:
2. override func tableView(\_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
3. performSegue(withIdentifier:"ShowBookList", sender: nil)
4. }
5. Run the app. It now looks like this:



صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

**Change the BookListTableViewController's Title**

1. In **BookListTableViewcontroller.swift,**add an instance variable:

var author: Author!

1. In **AuthorListTableViewController.swift,**update *tableView(didSelectRowAtIndexPath)*to the following:

override func tableView(\_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

let author = authors[indexPath.row]

performSegue(withIdentifier:"ShowBookList", sender: author)

}

This method has a  *sender*parameter that you previously set to *nil.*Now you'll use it to send along the *Author*object from the row that the user tapped on.

1. Add the *prepareForSegue(sender)*method to **AuthorListTableViewController.swift:**
2. override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
3. if segue.identifier == "ShowBookList" {
4. let bookListTableViewController = segue.destination as! BookListTableViewController
5. bookListTableViewController.author = sender as! Author
6. }
7. }

Inside the  *prepare(for segue, sender),*you need to give the **BookListTableViewController**the *Author*object from the row that the user tapped. That's why you put that object in the *sender*parameter earlier.

1. Run the app and notice that when you tap the row for an author, the next screen properly takes over the title. (The name of the chosen author now appears in the navigation bar.)

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

**Insert Books**

1. Refer to **Load & Delete**tab on how to create an *Add* bar button item to the Book List Table View Controller. Make sure you connect this button by **Ctrl + Drag**-ingit to the **BookListTableViewController.swift:**Connection: **Action**  
   Name:  **addBook**Type: **UIBarButtonItem**
2. Add these two instance variables to your **BookListTableViewController.swift:**
3. var author: Author!

var managedObjectContext = (UIApplication.shared.delegate as! AppDelegate).persistentContainer.viewContext

1. Copy and paste the codes below within the *addBook*method:
2. @IBAction func addBook(sender: UIBarButtonItem) {
3. let alert = UIAlertController(title: "New Book", message: "Add a new Book",
4. preferredStyle: .alert)
5. let newRowIndex = self.author.books?.count
6. let saveAction = UIAlertAction(title: "Save", style: .default) {
7. \_ in
8. let textField = alert.textFields![0]
9. let newBook = Book(context: self.managedObjectContext)
10. newBook.title = textField.text
11. self.author.addToBooks(newBook)
12. do {
13. try self.managedObjectContext.save()
14. print("Success!")
15. } catch {
16. print("Error: \(error)")
17. }
18. let indexPath = IndexPath(row: newRowIndex!, section: 0)
19. self.tableView.insertRows(at: [indexPath], with: .automatic)
20. }
21. let cancelAction = UIAlertAction(title: "Cancel", style: .default) {
22. UIAlertAction -> Void in
23. }
24. alert.addTextField {
25. UITextField -> Void in
26. }
27. alert.addAction(saveAction)
28. alert.addAction(cancelAction)
29. present(alert, animated: true, completion: nil)

}

The codes above should look familiar. The only difference is that we are storing  *books* through the Author object. Run your app and check whether the "Add Book" feature is working by creating book records. You will see a "Success!" message in Xcode's console if you followed the steps correctly.

**Display Related Books**

1. To count how many books there are to an Author:

author.books.count

Make sure you paste the code above within the  *tableView(numberOfRowsInsection)*method.

1. Copy and paste these codes to the *tableView(cellForRowAtIndexPath)*method to display each author's books:
2. override func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
3. let cell = tableView.dequeueReusableCell(withIdentifier: "Cell")!
4. let book = self.author.books?[indexPath.row] as! Book
5. cell.textLabel?.text = book.title
6. return cell

}

# Fetch Requests

So far, if we translated our CoreData transactions to SQL (Sequenced Querying Language), we'd see the bread and butter of SQL: "SELECT \* FROM table\_name", "INSERT INTO table\_name", and "DELETE FROM table\_name".

Next, we will cover the critical WHERE and ORDER BY clauses. Say for example, using SQL, we wanted to find all users in our records who lived in California. We might write something like "SELECT \* FROM users WHERE address LIKE %CA%".

It's useful to think about SQL syntax in dealing with the object model, but with CoreData Apple has given us convenient object oriented tools, and we will be using these to deal with our data. So let's get started learning our two new techniques: NSPredicate and NSSortDescriptor!

### NSPredicate

We attach an NSPredicate to our fetch request object to filter results in the same way that a SQL WHERE clause allows for. To do so, we create an instance of NSPredicate, passing a format string and a value (or multiple values):

let requestPredicate = NSPredicate(format: "someValue == %@", someValueVariable)

fetchRequest.predicate = requestPredicate

**This example shows one of the most basic uses of format strings: comparing equality. The %@ symbol within the format string represents where the value that we pass will be interpolated.**

**Format strings can take many more forms, and we will cover those in the next module.**

**Note: NSPredicate is not limited to just fetch requests; it can also be attached to arrays!**

### NSSortDescriptor

The NSSortDescriptor class correlates to SQL's ORDER clause. It has less options than NSPredicate, but the pattern remains the same. Here, we start by instantiating an object of the NSSortDescriptor class, specifying the key that we want to sort by and the order that we want the results sorted in:

let sortDescriptor = NSSortDescriptor(key: "createdAt", ascending: false)

fetchRequest.sortDescriptors = [sortDescriptor]

**Note:** the fetchRequest's sortDescriptor property expects an array of sort descriptors, so even if you have only one sort descriptor, you still have to place it inside of an array (like above).

**Note:** multiple sort descriptors take priority in the order they were added; if you add two descriptors: name ascending, and then date\_of\_birth ascending for example, you will see the following results: all of the people whose names begin with "A" will be at the top of the list, and they will be ordered by their birthday, then will come the people whose names start with "B", also ordered by their birthdays. Using this technique, we get some handy grouping functionality.

### All together now!

We can easily add multiple sortDescriptors, but what if we want to add multiple NSPredicates? There's special syntax for that: NSCompoundPredicate -- we just pass the logical operator AND or OR, and an array of the predicates we want to compound. Let's see what it looks like when we put all of the pieces together:

let fetchRequest:NSFetchRequest<Thing> = Thing.fetchRequest()

let firstPredicate = NSPredicate(format: "number >= %@", someValue")

let secondPredicate = NSPredicate(format: "name == %@", nameAttribute")

let firstSort = NSSortDescriptor(key: "locationStringAttribute", ascending:true)

let secondSort = NSSortDescriptor(key: "numberAttribute", ascending:true)

fetchRequest.predicate = NSCompoundPredicate(type: .and, subpredicates: [firstPredicate, secondPredicate])

fetchRequest.sortDescriptors = [firstSort, secondSort]

do {

&nbsp;try context.fetch(fetchRequest)

} catch {

print(error

}

You can see how this grants us a lot of power in specifying our queries. Remember to put the pieces together one at a time, and soon big complex queries will become a breeze!

# Predicates and Format Strings

An instance of NSPredicate takes two arguments: a format string, and values.

There are a ton of options for the format strings, and we will list out some of the most common ones here:

### "attributeName == %@"

Since we attach an NSPredicate to a fetchRequest of a specified entity, we can reference that entity's attribute names directly in format strings. In this case, we are providing a key, and filtering for direct comparison to a value, like so:

let personNamePredicate = NSPredicate(format: "name == %@", someName)

Basic operators go a long way in defining NSPredicates. For example, when comparing two dates, we can simply use a greater than or less than operator to signify before or after a certain date.

### "%K <= %@"

In addition to explicitly writing out the key name in our format string, we can specify the placeholder %K (that's a capital K!), and then pass a key before passing the value.

let dateCreatedPredicate = NSPredicate(format: "%K <= %@", "date", person1.date)

Also of note, these comparison operators that indicate measurements of amounts can be applied to either numbers OR dates.

### Logical Operators in Format Strings (&&, ||)

Furthermore, we can use && or || operators in our format strings to specify multiple pieces of logic. If we want to use multiple %@ symbols, we pass an array of arguments. Let's look at all of these ideas together:

let dateRangePredicate = NSPredicate(format: "date >= %@ && date <= %@", [date1, date2])

**Note that we have to specify 'date' on both sides of the AND/OR operator (this is pretty standard, but it's good to remember).**

### CONTAINS[c]

let fetchPredicate = NSPredicate(format: "title CONTAINS[c] %@", "F")

CONTAINS[c] gives us access to something like the SQL LIKE statement. In the example above, this predicate will filter out any strings that do not have the upper case letter: "F". Said another way, it finds all records that contain the letter "F" in their title attribute.