**>> Creating a TableView and storing the cell data using CoreData – Part B**

**A)Modeling Your Data**

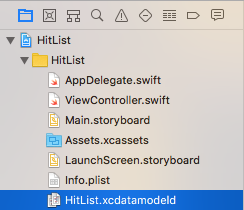
Now you know how to check for persistence, you can dive into Core Data. Your goal for the HitList app is simple: persist the names you enter so they’re available for viewing after a fresh app launch.

Up to this point, you’ve been using plain old Swift strings to store the names in memory. In this section, you’ll replace these strings with Core Data objects.

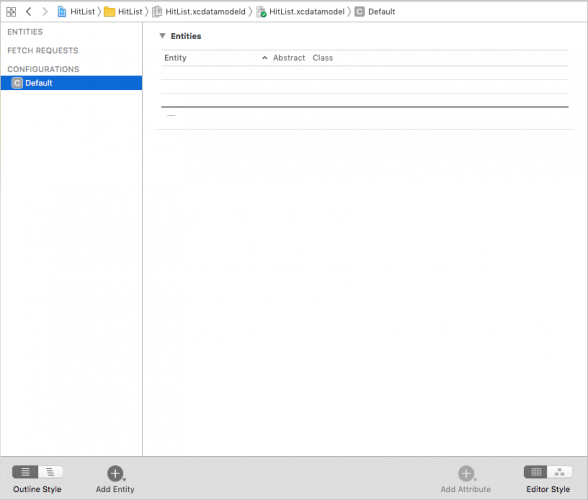
The first step is to create a **managed object model**, which describes the way Core Data represents data on disk.

By default, Core Data uses a SQLite database as the persistent store, so you can think of the Data Model as the database schema.

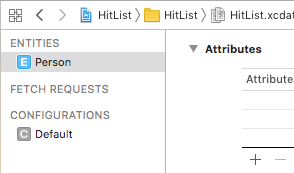
Since you’ve elected to use Core Data, Xcode automatically created a Data Model file for you and named it **HitList.xcdatamodeld**.



Open **HitList.xcdatamodeld**. As you can see, Xcode has a powerful Data Model editor:



Click on **Add Entity** on the lower-left to create a new entity. Double-click the new entity and change its name to **Person**, like so:



**Some terms in Core Data**

* An **entity** is a class definition in Core Data. The classic example is an Employee or a Company. In a relational database, an entity corresponds to a table.
* An **attribute** is a piece of information attached to a particular entity. For example, an Employee entity could have attributes for the employee’s name, position and salary. In a database, an attribute corresponds to a particular field in a table.
* A **relationship** is a link between multiple entities. In Core Data, relationships between two entities are called **to-one relationships**, while those between one and many entities are called **to-many relationships**. For example, a Manager can have a **to-many relationship** with a set of employees, whereas an individual Employee will usually have a **to-one relationship** with his manager.
* Now you know what an attribute is, you can add an attribute to Person object created earlier. Still in **HitList.xcdatamodeld**, select Person on the left-hand side and click the plus sign (+) under **Attributes**.
* Set the new attribute’s name to, er, **name** and change its type to **String**:

## B) Saving to Core Data

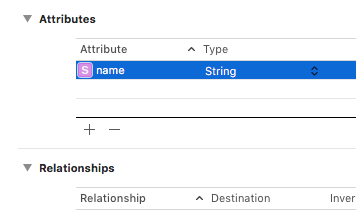
Open **ViewController.swift**, add the following Core Data module import below the UIKit import:

**Import CoreData**

This import is all you need to start using the Core Data API in your code.

Next, replace the names property definition with the following:

var people: [NSManagedObject]= [ ]

* 
* In Core Data, an attribute can be of one of several data types.

// MARK: - UITableViewDataSource

extension ViewController: UITableViewDataSource {

func tableView(\_ tableView: UITableView,

numberOfRowsInSection section: Int) -> Int {

return people.count

}

func tableView(\_ tableView: UITableView,

cellForRowAt indexPath: IndexPath)

-> UITableViewCell {

let person = people[indexPath.row]

let cell =

tableView.dequeueReusableCell(withIdentifier: "Cell",

for: indexPath)

cell.textLabel?.text =

person.value(forKeyPath: "name") as? String

return cell

}

}

The most significant change to these methods occurs in tableView(\_:cellForRowAt:). Instead of matching cells with the corresponding string in the model array, you now match cells with the corresponding NSManagedObject.

Note how you grab the name attribute from the NSManagedObject. It happens here:

cell.textLabel?.text = person.value(forKeyPath: "name") as? String

Next, find addName(\_:) and replace the save UIAlertAction with the following:

let saveAction = UIAlertAction(title: "Save", style: .default) {

[unowned self] action in

guard let textField = alert.textFields?.first,

let nameToSave = textField.text else {

return

}

self.save(name: nameToSave)

self.tableView.reloadData()

}

This takes the text in the text field and passes it over to a new method named save(name:). Xcode complains because save(name:) doesn’t exist yet. Add it below addName(\_:):

func save(name: String) {

guard let appDelegate =

UIApplication.shared.delegate as? AppDelegate else {

return

}

// 1

let managedContext =

appDelegate.persistentContainer.viewContext

// 2

let entity =

NSEntityDescription.entity(forEntityName: "Person",

in: managedContext)!

let person = NSManagedObject(entity: entity,

insertInto: managedContext)

// 3

person.setValue(name, forKeyPath: "name")

// 4

do {

try managedContext.save()

people.append(person)

} catch let error as NSError {

print("Could not save. \(error), \(error.userInfo)")

}

}

***NOTES: This is where Core Data kicks in! Here’s what the code does:***

1. Before you can save or retrieve anything from your Core Data store, you first need to get your hands on an NSManagedObjectContext. You can consider a managed object context as an in-memory “scratchpad” for working with managed objects.

Think of saving a new managed object to Core Data as a two-step process: first, you insert a new managed object into a managed object context; once you’re happy, you “commit” the changes in your managed object context to save it to disk.

Xcode has already generated a managed object context as part of the new project’s template. Remember, this only happens if you check the **Use Core Data** checkbox at the beginning. This default managed object context lives as a property of the NSPersistentContainer in the application delegate. To access it, you first get a reference to the app delegate.

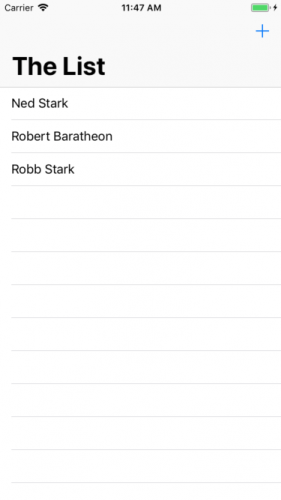
1. You create a new managed object and insert it into the managed object context. You can do this in one step with NSManagedObject’s static method: entity(forEntityName:in:).

You may be wondering what an NSEntityDescription is all about. Recall earlier, NSManagedObject was called a shape-shifter class because it can represent any entity. An entity description is the piece linking the entity definition from your Data Model with an instance of NSManagedObject at runtime.

1. With an NSManagedObject in hand, you set the name attribute using key-value coding. You must spell the KVC key (name in this case) **exactly** as it appears in your Data Model, otherwise, your app will crash at runtime.
2. You commit your changes to person and save to disk by calling save on the managed object context. Note save can throw an error, which is why you call it using the try keyword within a do-catch block. Finally, insert the new managed object into the people array so it shows up when the table view reloads.

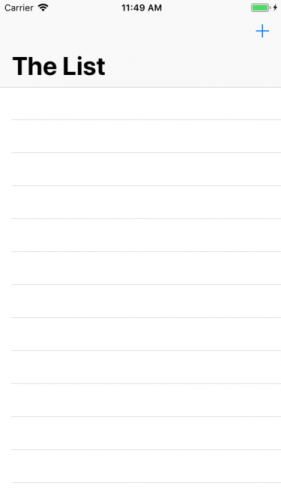
That’s a little more complicated than using an array of strings, but not too bad. Some of the code here, such as getting the managed object context and entity, could be done just once in your own init() or viewDidLoad() then reused later. For simplicity, you’re doing it all in the same method.

Build and run the app, and add a few names to the table view:



If the names are actually stored in Core Data, the HitList app should pass the persistence test. With the app in the foreground, go to the fast app switcher and then terminate it.

From Springboard, tap the HitList app to trigger a fresh launch. Wait, what happened? The table view is empty:



## C) Fetching from Core Data

To get data from your persistent store into the managed object context, you have to **fetch** it. Open **ViewController.swift** and add the following below viewDidLoad():

override func viewWillAppear(\_ animated: Bool) {

super.viewWillAppear(animated)

//1

guard let appDelegate =

UIApplication.shared.delegate as? AppDelegate else {

return

}

let managedContext =

appDelegate.persistentContainer.viewContext

//2

let fetchRequest =

NSFetchRequest<NSManagedObject>(entityName: "Person")

//3

do {

people = try managedContext.fetch(fetchRequest)

} catch let error as NSError {

print("Could not fetch. \(error), \(error.userInfo)")

}

}

Step by step, this is what the code does:

1. Before you can do anything with Core Data, you need a managed object context. Fetching is no different! Like before, you pull up the application delegate and grab a reference to its persistent container to get your hands on its NSManagedObjectContext.
2. As the name suggests, NSFetchRequest is the class responsible for fetching from Core Data. Fetch requests are both powerful and flexible. You can use fetch requests to fetch a set of objects meeting the provided criteria (i.e. give me all employees living in Wisconsin and have been with the company at least three years), individual values (i.e. give me the longest name in the database) and more.

Fetch requests have several qualifiers used to refine the set of results returned. For now, you should know NSEntityDescription is one of these required qualifiers.

Setting a fetch request’s entity property, or alternatively initializing it with init(entityName:), fetches *all* objects of a particular entity. This is what you do here to fetch all Person entities. Also note NSFetchRequest is a generic type. This use of generics specifies a fetch request’s *expected* return type, in this case NSManagedObject.

1. You hand the fetch request over to the managed object context to do the heavy lifting. fetch(\_:) returns an array of managed objects meeting the criteria specified by the fetch request.

Build and run the application. Immediately, you should see the list of names you added earlier:

