Chapter 19
Working with Core Data

> Learn not to add too many features right away, and get the core idea built and tested.
>
> – Leah Culver

Congratulations on making it this far! By now you've already built a simple app for users to list their favorite restaurants. If you've worked on the previous exercise, you should understand the fundamentals of how to add a restaurant; I've tried to keep thing simple and focus on the basics of UITableView. Up to this point, all restaurants have been predefined in the source code and stored in an array. If you want to save a restaurant, the simplest way is to add a new restaurant to the existing restaurants array.

However, if you do it this way, you can't save the new restaurant permanently. Data holding in memory (e.g. array) is volatile. Once you quit the app, all the changes are gone. We need to find a way to save the data in a persistent manner.

To save the data permanently, we'll need to save in a persistent storage-like file or database. By saving the data to a database, for example, the data will be safe even if the app quits or crashes. Files are another way to save data, but they are more suitable for storing small amounts of data that do not require frequent changes. For instance, files are commonly used for storing application settings. If you open the *Support Files* folder in project navigator, you'll find the Info.plist file. This property file is used for storing your project settings.

The FoodPin app may need to store thousands of restaurant records. Users may also add or remove the restaurant records quite frequently. In this case, a database is a suitable way to handle a large set of data. In this chapter, I will walk you through the Core Data framework and show you how to use it to manage data in database. You will make a lot of changes to your existing FoodPin project, but after going through this chapter your app will allow users to save their favorite restaurants persistently. If you haven't done the previous exercise, you can download this project's templates (https://www.dropbox.com/s/hy9j6v0q0tmv5fr/FoodPinStaticTableViewExercise.zip?dl=0)

to start with.

What is Core Data?

When we talk about persistent data, you probably think of databases. If you are familiar with Oracle or MySQL, you know that a relational database stores data in the form of tables, rows and columns; your app talks to the database by using an SQL (Structured Query Language) query. However, don't mix up Core Data with databases. Though SQLite database is the default persistent store for Core Data on iOS, Core Data is not exactly a relational database - it is actually a framework that lets developers interact with database (or other persistent storage) in an object-oriented way.

Take the FoodPin app as an example. If you want to save the data to database, you are responsible for writing the code to connect to the database and retrieve or update the data using SQL. This would be a burden for developers, especially for those who do not know SQL.

Core Data provides a simpler way to save data to a persistent store of your choice. You can map the objects in your apps to the table in the database. Simply put it allows you to manage records (select/insert/update/delete) in the database without even knowing any SQL.

Core Data Stack

Before we start working on the project, you need to first have a basic understanding of the Core Data Stack (see figure 19-1):

- **Managed Object Context** – Think of it as a *scratch pad* or temporary memory area containing objects that interact with data in persistent store. Its job is to manage objects created and returned using Core Data framework. Among the components in the Core Data stack, the managed object context is the one you'll work directly with most of the time. In general, whenever you need to fetch and save objects in the persistent store, the context is the first component you'll interact with.
- **Persistent Store Coordinator** – SQLite is the default persistent store in iOS. However, Core Data allows developers to set up multiple stores containing different entities. The persistent store coordinator is the party responsible for managing different persistent object stores and saving the objects to the stores. Forget about it if you don't understand

what it is; you won't interact with the persistent store coordinator directly when using Core Data.

- **Managed Object Model** – This describes the schema that you use in the app. If you have some background in databases, think of this as the database schema. However, the schema is represented by a collection of objects (also known as entities). For example, a collection of model objects can be used to represent the collection of restaurants in the FoodPin app. In Xcode, the managed object model is defined in a file with the extension `.xcdatamodeld`. You can use the visual editor to define the entities and their attributes and relationships.
- **Persistent Store** - This is the repository in which your data is actually stored. Usually it's a database, and SQLite is the default database. But it can also be a binary or XML file.

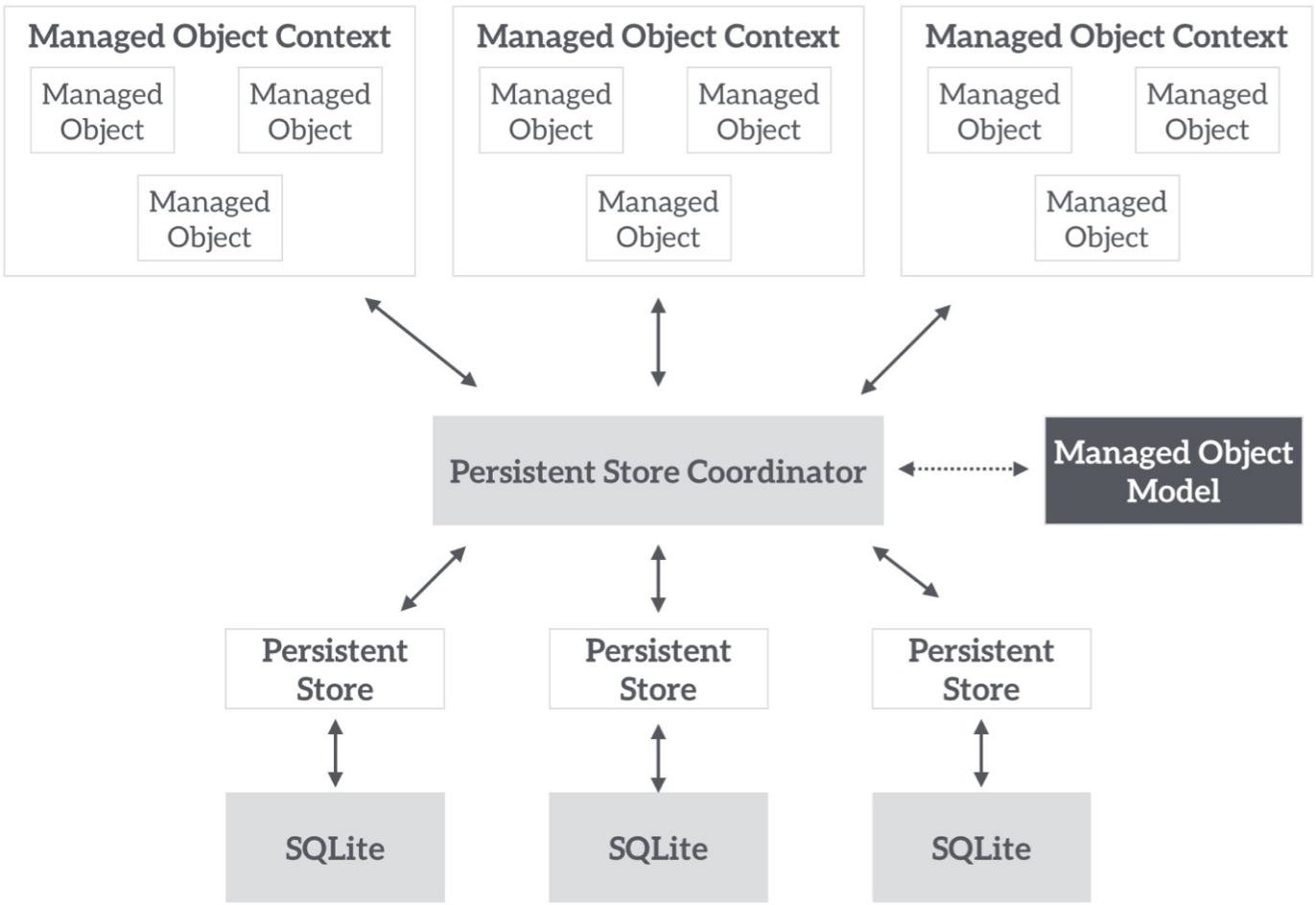

Figure 19-1. Core Data Stack

If you don't completely understand the terms or the Core Data stack yet, don't worry. You will

learn along the way as we convert the FoodPin app from arrays to Core Data.

Using Core Data Template

The simplest way to use Core Data is to enable the Core Data option during project creation. Xcode will generate the required code in `AppDelegate.swift` and create the data model file for Core Data.

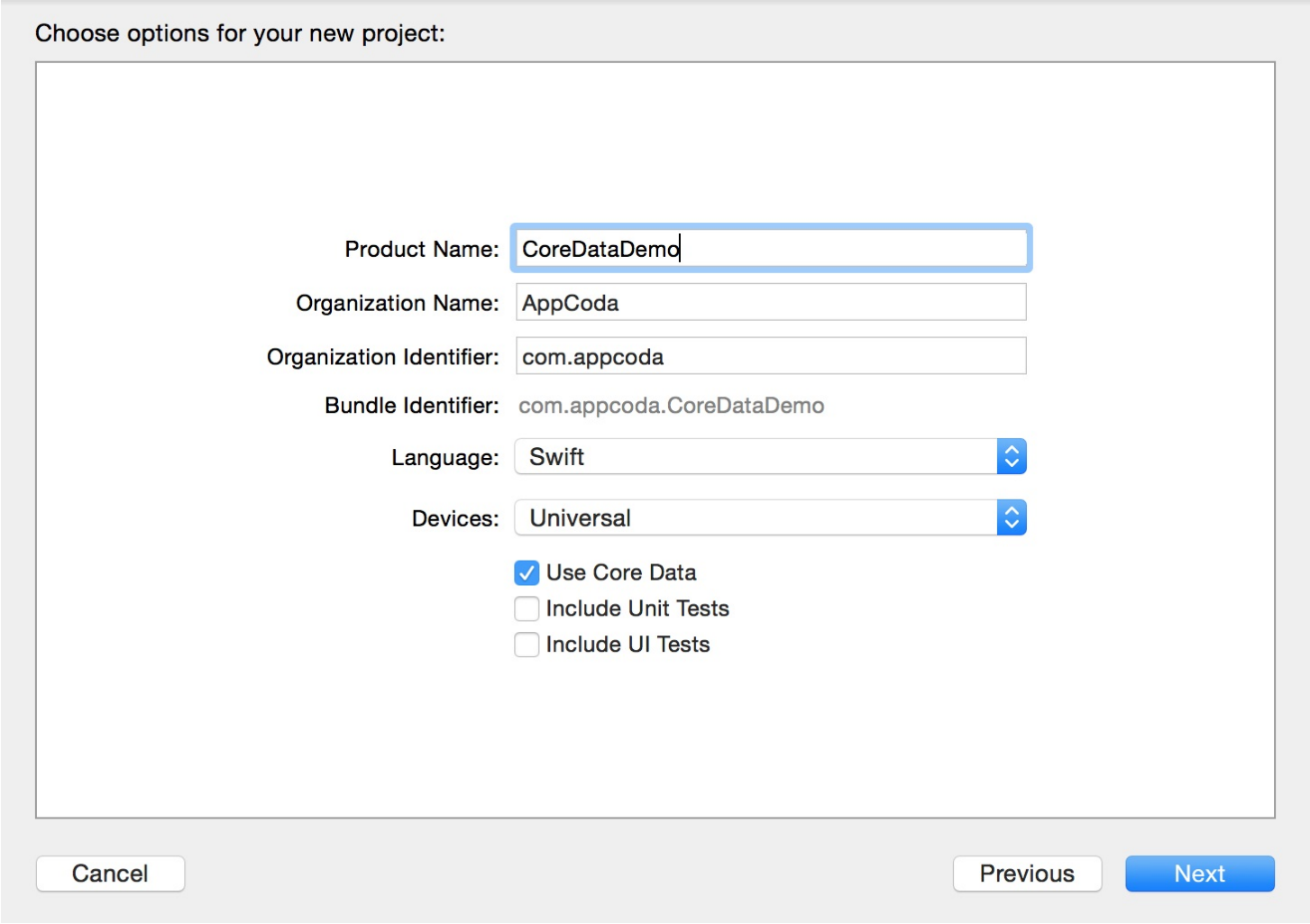

Figure 19-2. Enable Core Data option

If you create a *CoreDataDemo* project with Core Data option enabled, you will see the following variables and method generated in the `AppDelegate` class:

```
// MARK: - Core Data stack
```

```
lazy var applicationDocumentsDirectory: NSURL = {
    // The directory the application uses to store the Core Data store file.
This code uses a directory named "com.appcoda.CoreDataDemo" in the
application's documents Application Support directory.
    let urls =
NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inDomains:
.UserDomainMask)
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not
optional. It is a fatal error for the application not to be able to find and
load its model.
    let modelURL = NSBundle.mainBundle().URLForResource("CoreDataDemo",
withExtension: "momd")!
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This
implementation creates and returns a coordinator, having added the store for
the application to it. This property is optional since there are legitimate
error conditions that could cause the creation of the store to fail.
    // Create the coordinator and store
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel:
self.managedObjectModel)
    let url =
self.applicationDocumentsDirectory.URLByAppendingPathComponent("SingleViewCoreDa

    var failureReason = "There was an error creating or loading the
application's saved data."
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
configuration: nil, URL: url, options: nil)
    } catch {
        // Report any error we got.
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the
application's saved data"
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

        dict[NSUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999,
userInfo: dict)
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and
terminate. You should not use this function in a shipping application, although
```

```
it may be useful during development.
        NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }

    return coordinator
}()

lazy var managedObjectContext: NSManagedObjectContext = {
    // Returns the managed object context for the application (which is already
bound to the persistent store coordinator for the application.) This property
is optional since there are legitimate error conditions that could cause the
creation of the context to fail.
    let coordinator = self.persistentStoreCoordinator
    var managedObjectContext = NSManagedObjectContext(concurrencyType:
.MainQueueConcurrencyType)
    managedObjectContext.persistentStoreCoordinator = coordinator
    return managedObjectContext
}()

// MARK: - Core Data Saving support

func saveContext () {
    if managedObjectContext.hasChanges {
        do {
            try managedObjectContext.save()
        } catch {
            // Replace this implementation with code to handle the error
appropriately.
            // abort() causes the application to generate a crash log and
terminate. You should not use this function in a shipping application, although
it may be useful during development.
            let nserror = error as NSError
            NSLog("Unresolved error \(nserror), \(nserror.userInfo)")
            abort()
        }
    }
}
```

The variables allow you to easily access the Core Data stack such as the managed object context. The question is how we can use this code template in our existing Xcode project. You can simply copy and paste the code into the `AppDelegate.swift` file of your Food Pin project, but you will need to implement a couple of changes.

```
let modelURL = NSBundle.mainBundle().URLForResource("CoreDataDemo",
withExtension: "momd")!
```

```
let url =
self.applicationDocumentsDirectory.URLByAppendingPathComponent("SingleViewCoreDa
```

The original code template was generated for the `CoreDataDemo` project. Xcode names the SQLite and data model file using the project name. For the FoodPin project, we use the name *FoodPin* for both files. So change the above two lines to the following:

```
let modelURL = NSBundle.mainBundle().URLForResource("FoodPin", withExtension:
"momd")!

let url =
self.applicationDocumentsDirectory.URLByAppendingPathComponent("FoodPin.sqlite")
```

Finally, add the following `import` statement at the beginning of the `AppDelegate` class to import the Core Data framework:

```
import CoreData
```

Quick note: For reference, you can also download this project template (https://www.dropbox.com/s/8s9c49wdb28hf3e/FoodPinCoreDataTemplate.zip?dl=0) to continue.

Creating the Managed Object Model

Now that you've prepared the code for accessing the Core Data stack, let's move on to create the managed object model. In the project navigator, right-click the `FoodPin` folder and select `New File...`. Choose Core Data and select Data Model.

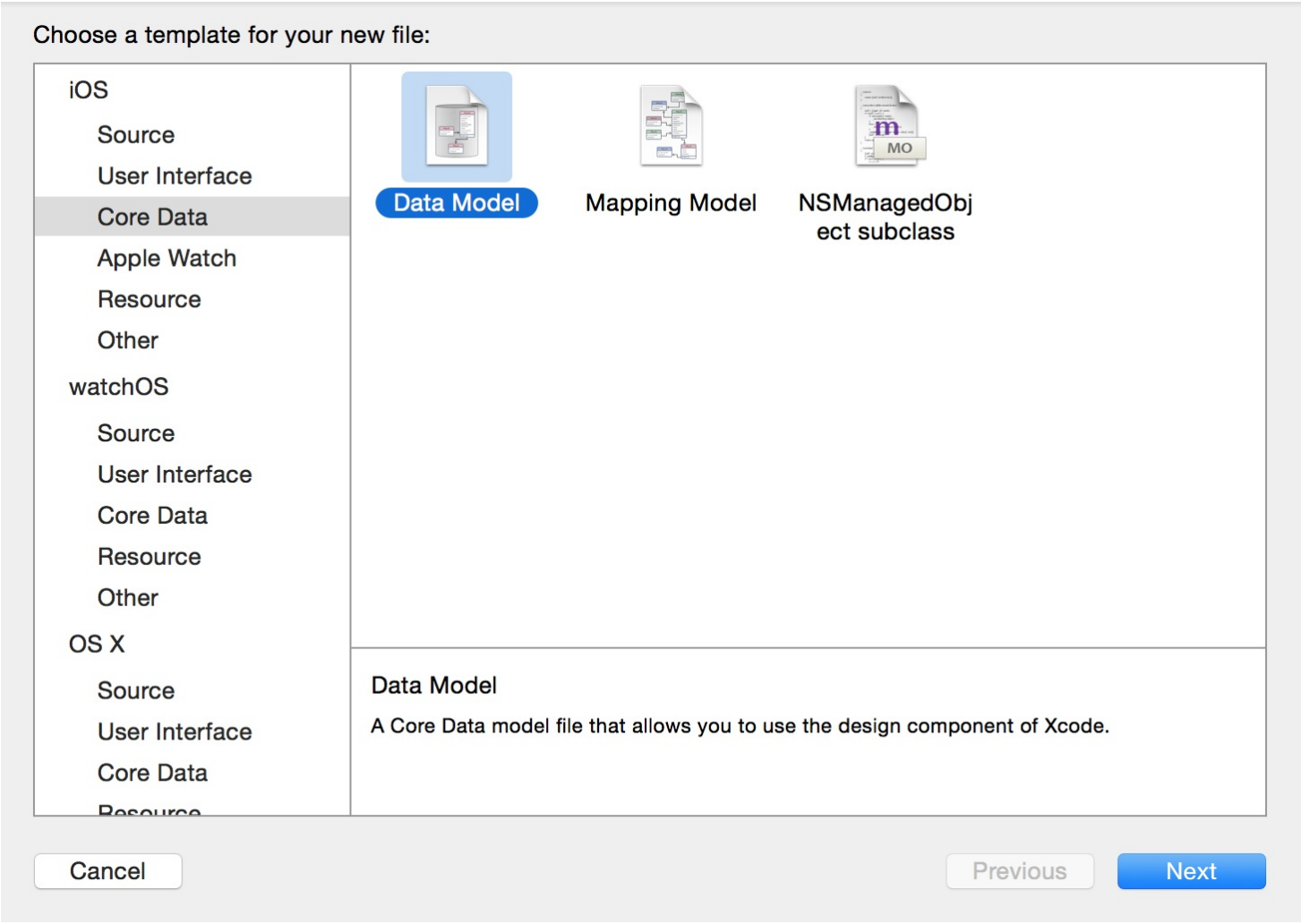

Figure 19-3. Creating the data model using Data Model template

Name the model `FoodPin` and click `Create` to create the data model. Once created, you should find a file named `FoodPin.xcdatamodeld` in the project navigator. Select it to open the data model editor. From here, you can create entities for the managed object model.

As we would like to store the `Restaurant` object in database, we will create a `Restaurant` entity that matches the `Restaurant` class in our code. To create an entity, click the *Add Entity* button at the bottom of the editor pane and name the entity `Restaurant`.

In order to save the data from the `Restaurant` object to the database, we will add several attributes for the entity that align with the attributes of the object. Simply click the `+` button under the attributes section to create a new attribute. Add six attributes for the `Restaurant` entity including name, type, location, image, isVisited, and rating. Refer to figure 19-4 for

details.

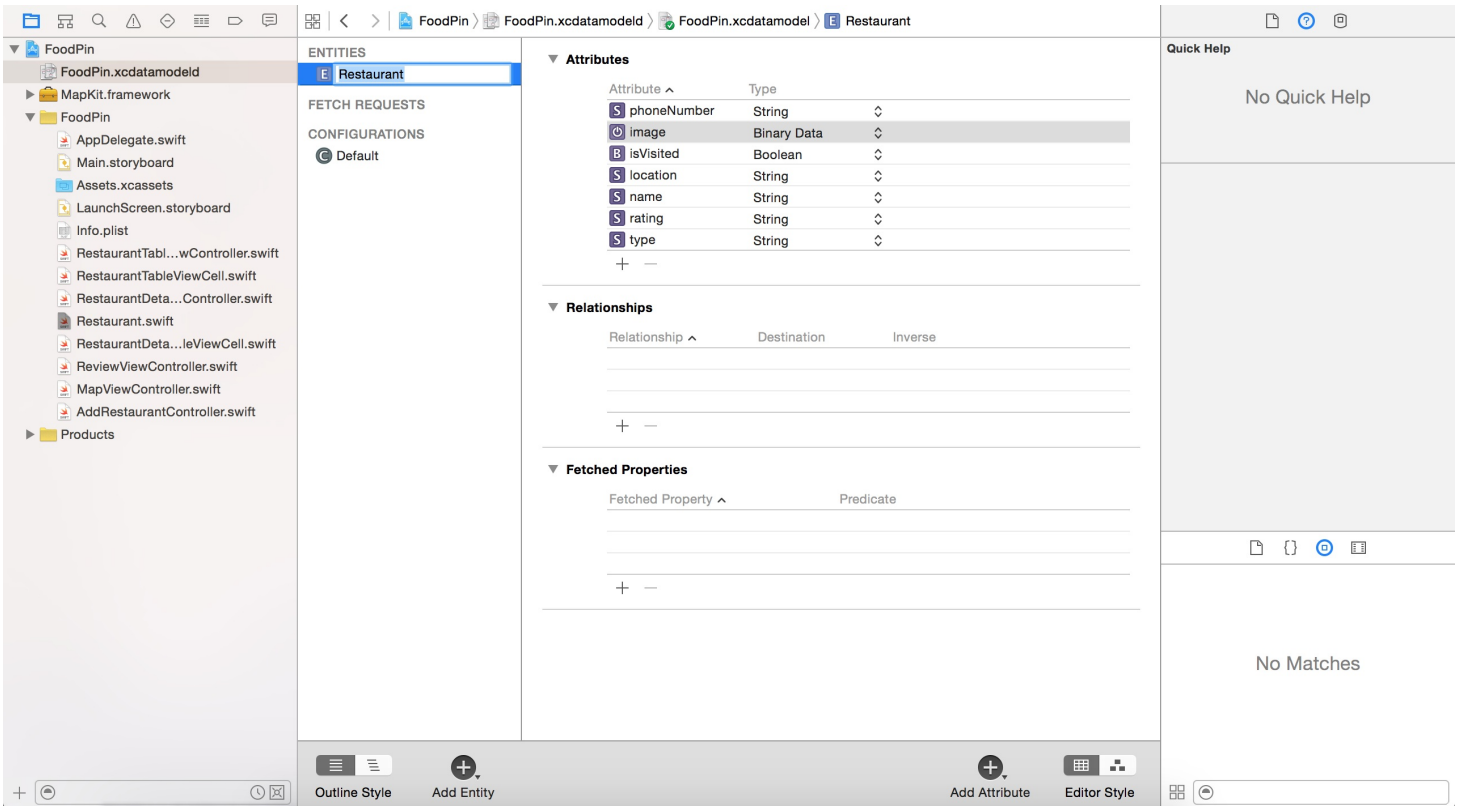

Figure 19-4. Adding attributes to the Restaurant entity

The attribute types of *name*, *type*, *location*, *phoneNumber*, *isVisited*, and *rating* are trivial, but why do we set the attribute type of *image* to Binary Data?

Presently, the restaurant images are bundled in the app, and managed by the asset catalog. This is why we can load an image by passing `UIImage` with the set name. When user creates a new restaurant, the image is loaded from an external source whether it's from the built-in photo library or taken from camera. In this case, we can't just store the file name. Instead, we save the actual data of the image into database. Binary data type is used for this purpose.

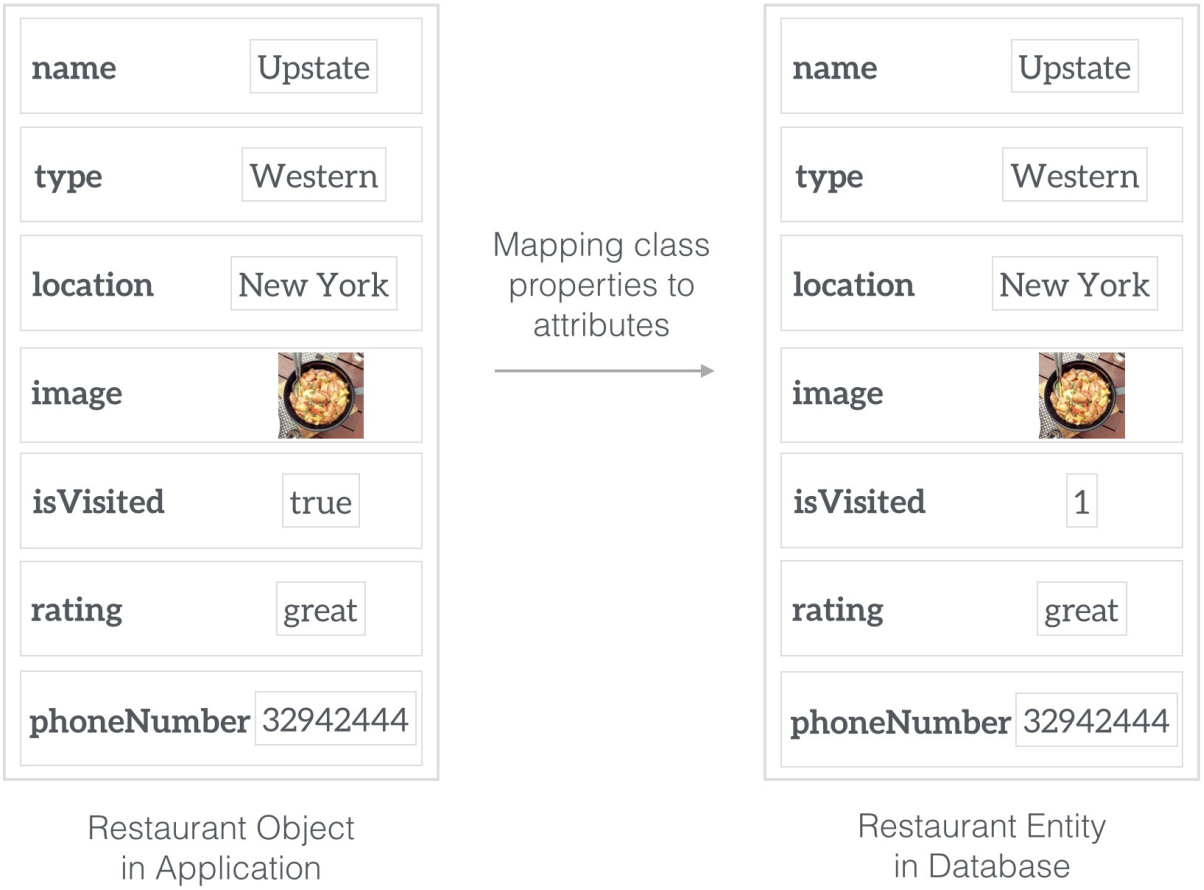

Figure 19-5. Sample relationship between model object and entity

You can configure the properties of an attribute in the Data Model inspector. For example, the *name* attribute is a required attribute. You can uncheck the *Optional* checkbox to make it mandatory. For the FoodPin project, you can set the *name*, *type*, and*location* as required.

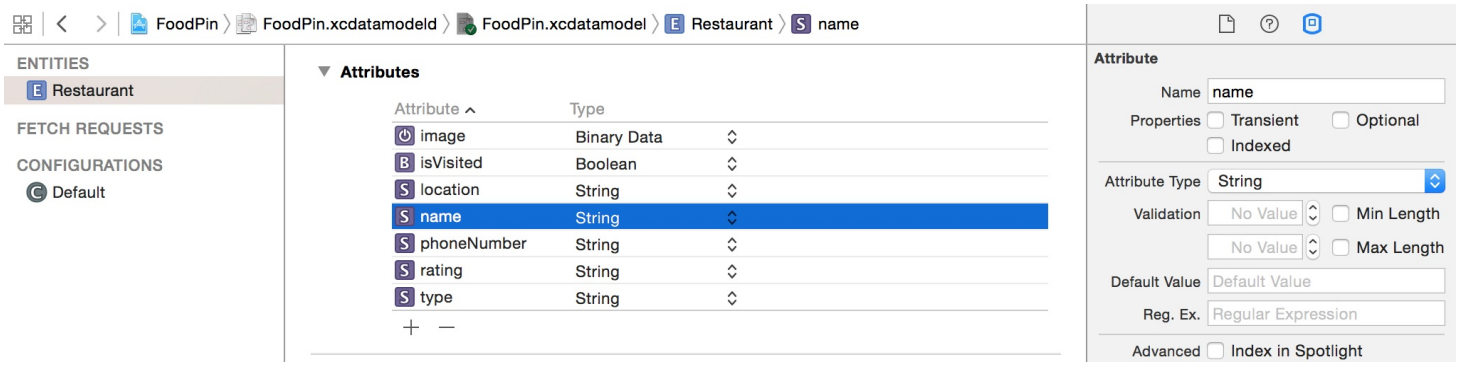

Figure 19-6. Editing attribute properties in the Data Model inspector

Creating Managed Objects

Model objects that tie into in the Core Data framework are known as *managed objects*. Managed objects are at the heart of any Core Data application. Now that you've created the managed object model, the next thing is to create the managed objects. For the FoodPin project, you can convert the `Restaurant` class to a managed object class by replacing the original code in `Restaurant.swift` like this:

```
import Foundation
import CoreData

class Restaurant:NSManagedObject {
    @NSManaged var name:String
    @NSManaged var type:String
    @NSManaged var location:String
    @NSManaged var phoneNumber:String?
    @NSManaged var image:NSData?
    @NSManaged var isVisited:NSNumber?
    @NSManaged var rating:String?
}
```

Managed object is a subclass of `NSManagedObject` which represents an entity. In Swift, we add the `@NSManaged` keyword before each property definition that corresponds to the attribute of the `Restaurant` entity. As you can see, an optional attribute has a corresponding optional property (e.g. phoneNumber) in the class.

Core Data has support for many common data types, such as String. For *binary data*, the corresponding data type in managed object is `NSData`. However, Core Data does not have Boolean type for managed objects; if you need to use Boolean type, use `NSNumber` instead. `NSNumber` uses a non-zero value to represent true, while a zero value means false.

Once you've converted the `Restaurant` class to a managed object, you have to assign it to the `Restaurant` entity in order to establish a relationship between them. Select the `Restaurant` entity in `FoodPin.xcdatamodeld`. In the Data Model inspector, set the *class* option to `Restaurant` and *module* to `Current Product Module`.

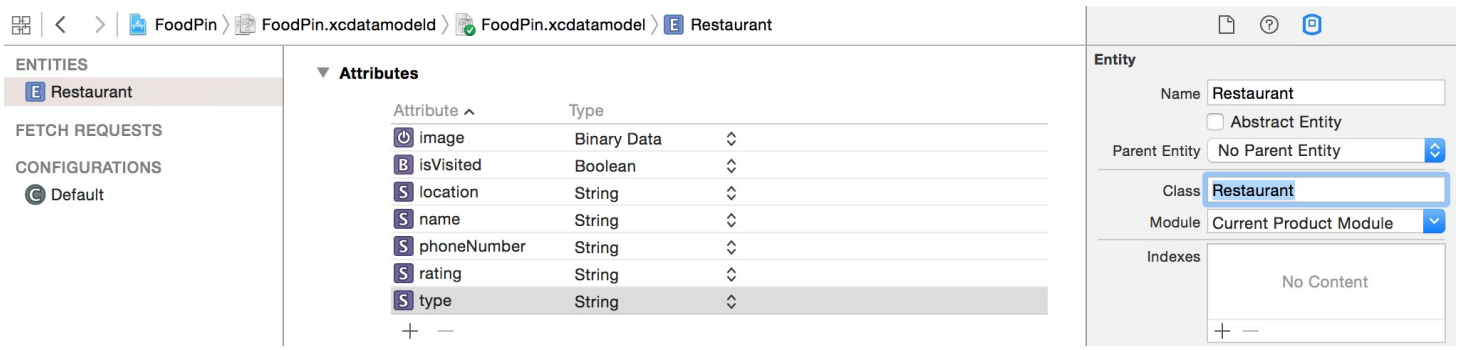

Figure 19-7. Assign the entity with the Restaurant class

When using a Swift subclass of the NSManagedObject class, it is a must to prefix the class name in the Class field with the name of your module.

Now the managed object is created. However, due to the change of the `image` and `isVisited` properties, Xcode detects multiple errors in your project. To fix the errors, you have to go through a number of changes. First, in `RestaurantTableViewController.swift`, you no longer use file name to load an image.

```
cell.thumbnailImageView.image = UIImage(named:
restaurants[indexPath.row].image)
```

The image is now stored as an `NSData` object. To load the image, instead of passing the image through the `named` parameter, initialize the `UIImage` object using the `data` parameter:

```
cell.thumbnailImageView.image = UIImage(data:
restaurants[indexPath.row].image!)
```

The same applies to the `MapViewController` class. So replace the following line of code:

```
leftIconView.image = UIImage(named: restaurant.image)
```

with:

```
leftIconView.image = UIImage(data: restaurant.image!)
```

Furthermore, the same error happens in `RestaurantDetailViewController.swift`. Again, replace the image instantiation with the `data` parameter:

```
restaurantImageView.image = UIImage(data: restaurant.image!)
```

The other error is related to the `isVisited` property. Xcode should indicate the following line in `RestaurantTableViewController.swift` with an error:

```
cell.accessoryType = restaurants[indexPath.row].isVisited ? .Checkmark : .None
```

The `isVisited` property is no longer in `Bool` type; it is now in `NSNumber` type. You need to modify the code to get the Boolean value by accessing the `boolValue` property:

```
if let isVisited = restaurants[indexPath.row].isVisited?.boolValue {
    cell.accessoryType = isVisited ? .Checkmark : .None
}
```

There is another error related to the `isVisited` variable in `RestaurantDetailViewController.swift`:

```
cell.valueLabel.text = (restaurant.isVisited) ? "Yes, I've been here before" :
"No"
```

Again, you need to access the `boolValue` property to get the Boolean value:

```
if let isVisited = restaurant.isVisited?.boolValue {
    cell.valueLabel.text = isVisited ? "Yes, I've been here before" : "No"
}
```

Because the rating property is changed to an optional, we have to update these lines of code in `RestaurantDetailViewController.swift` from:

```
if restaurant.rating != "" {
    ratingButton.setImage(UIImage(named: restaurant.rating), forState:
UIControlState.Normal)
}
```

to:

```
if let rating = restaurant.rating where rating != "" {
    ratingButton.setImage(UIImage(named: restaurant.rating!), forState:
UIControlState.Normal)
}
```

The last error is related to the `restaurants` array in the `RestaurantTableViewController` class. As we're going to store the restaurants in database, simply declare an empty array:

```
var restaurants:[Restaurant] = []
```

Quick note: I encourage you to fix the errors on your own. But for reference, you can download the Xcode project from https://www.dropbox.com/s/ro6rlr2k0g8bf5k/FoodPinCoreDataTemplate2.zip?dl=0.

As there is no restaurant data, your app should now display a blank table when launched. Next up, we'll implement the AddTableViewController class and save the new restaurant to database.

Working with Managed Objects

To save a restaurant into database through Core Data, you will have to insert a `Restaurant` object like this:

1. Get the managed object context from `AppDelegate`. In iOS SDK, you can use `UIApplication.sharedApplication().delegate as? AppDelegate` to get the `AppDelegate` object.

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

1. Create a managed object for the `Restaurant` entity by calling `insertNewObjectForEntityForName`.

```
NSEntityDescription.insertNewObjectForEntityForName("Restaurant",
inManagedObjectContext: managedObjectContext) as! Restaurant
```

1. Lastly, tell the context to save the new object into the database by calling the `save` method.

```
do {
    try managedObjectContext.save()
} catch {
    print(error)
    return
}
```

Quick note: In Swift 2, it comes with an exception-like model using try-throw-catch keywords. You use do-catch statement to catch errors and handle them accordingly. As you may notice, we put a try keyword in front of the method call. With the introduction of the new error handling model in Swift 2.0, some methods can throw errors to indicate failures. When we invoke a throwing method, you will need to put a try keyword in front of it. For details of error handling, please refer to the appendix.

As you can see, Core Data shields you away from the underlying logics of database management. You do not need to understand how to insert a record into the database using SQL. All is done by using the Core Data APIs.

Saving a New Restaurant to the Database

Now that you have some ideas about working with managed objects, let's update the `AddTableViewController` class to save a new restaurant to the database.

First, add the following `import` statement at the very beginning of `AddTableViewController.swift` so that the class can utilize the Core Data framework:

```
import CoreData
```

Declare a restaurant variable in the `AddTableViewController` class:

```
var restaurant:Restaurant!
```

In the `save` method, apply what you've just learned to save the `restaurant` object into the persistent store. Insert the following code after the form validation:

```
if let managedObjectContext = (UIApplication.sharedApplication().delegate as?
AppDelegate)?.managedObjectContext {
    restaurant =
NSEntityDescription.insertNewObjectForEntityForName("Restaurant",
inManagedObjectContext: managedObjectContext) as! Restaurant
    restaurant.name = name!
    restaurant.type = type!
    restaurant.location = location!
    if let restaurantImage = imageView.image {
        restaurant.image = UIImagePNGRepresentation(restaurantImage)
    }
    restaurant.isVisited = isVisited
```

```
    do {
        try managedObjectContext.save()
    } catch {
        print(error)
        return
    }
}
```

The above code is pretty much the same as what we discussed in the previous section. However, there are two lines of code that are new to you. First, it's related to the `image` property.

```
restaurant.image = UIImagePNGRepresentation(restaurantImage)
```

As we've changed the type of the `image` property from `String` to `NSData`, we have to retrieve the data of the selected image. The UIKit framework provides a set of built-in functions for graphics operations. The `UIImagePNGRepresentation` function allows us to get the data of a specified image in PNG format.

The `restaurant.isVisited` property is currently in `NSNumber` type. In Swift, it automatically bridges certain native number types including Int, UInt, Float, Double, and Bool to NSNumber. This is why the `isVisited` (in Bool) can be assigned to `restaurant.isVisited` (in NSNumber) directly.

```
restaurant.isVisited = isVisited
```

If you run the app now and save a new restaurant, the app should be able to save the record into database without any errors. However, your app is not ready to display the restaurant just added. That's what we're going to do next.

Fetching Data Using Core Data

To fetch data using Core Data, the simplest way is to use a convenient method provided by the managed object context:

```
if let managedObjectContext = (UIApplication.sharedApplication().delegate as?
AppDelegate)?.managedObjectContext {
    let fetchRequest = NSFetchRequest(entityName: "Restaurant")
```

```
    do {
        restaurants = try
managedObjectContext.executeFetchRequest(fetchRequest) as! [Restaurant]
        tableView.reloadData()
    } catch {
        print(error)
    }
}
```

By initializing an `NSFetchRequest` with the specified entity, you call the `executeFetchRequest` method to get an array of `Restaurant` objects. Then you reload the table view to display the restaurant on screen. Apparently, you can put the above code in the `viewWillAppear` method to load the latest restaurant objects from database.

However, we're not going to use the convenient method for fetching the records. Instead, I'll introduce you another API called `NSFetchedResultsController`. You may wonder why we don't just use the simple method - the primary reason is for performance. Whether you add a new record or remove a record from database, we load all restaurant records from database and re-display them in the table view. That's not an efficient way to manage the data. A better way to do that is like this:

- When adding a new record, we add a new row in the table view.
- When removing a record, we just need to remove a row from the table view.

The fetched results controller is specially designed for managing the results returned from a Core Data fetch request, and providing data for a table view. It monitors changes to objects in the managed object context and reports changes in the result set to its delegate.

Let's see how to use `NSFetchedResultsController` to retrieve the restaurants.

In `RestaurantTableViewController.swift`, first import the CoreData framework:

```
import CoreData
```

Then adopt the `NSFetchedResultsControllerDelegate` protocol:

```
class RestaurantTableViewController:UITableViewController,
NSFetchedResultsControllerDelegate
```

The `NSFetchedResultsControllerDelegate` protocol provides methods to notify its delegate whenever there are any changes in the controller's fetch results. Later we'll implement the methods. For now, declare an instance variable for the fetched results controller:

```
var fetchResultController:NSFetchedResultsController!
```

And add the following code in the `viewDidLoad` method:

```
let fetchRequest = NSFetchRequest(entityName: "Restaurant")
let sortDescriptor = NSSortDescriptor(key: "name", ascending: true)
fetchRequest.sortDescriptors = [sortDescriptor]

if let managedObjectContext = (UIApplication.sharedApplication().delegate as?
AppDelegate)?.managedObjectContext {

    fetchResultController = NSFetchedResultsController(fetchRequest:
fetchRequest, managedObjectContext: managedObjectContext, sectionNameKeyPath:
nil, cacheName: nil)
    fetchResultController.delegate = self

    do {
        try fetchResultController.performFetch()
        restaurants = fetchResultController.fetchedObjects as! [Restaurant]
    } catch {
        print(error)
    }
}
```

We first create an `NSFetchRequest` object with the `Restaurant` entity and specify the sort order using an `NSSortDescriptor` object. `NSSortDescriptor` lets you describe how the fetched objects are sorted. Here we specify that the `Restaurant` objects should be sorted in ascending order using the *name* key.

After creating the fetch request, we initialize a fetched result controller and specify its delegate for monitoring data changes. Lastly, we call the `performFetch()` method to execute the fetch request. When complete, we get the `Restaurant` objects by accessing the `fetchedObjects` property. The `fetchedObjects` property returns an array of `AnyObject`, so we have to cast it to `[Restaurant]`.

If you compile and run the app now, it should display the restaurants that you previously added. However, if you try to add another new restaurant, the table does not load the new

record.

There are still something left.

When there is any content change, the following methods of the `NSFetchedResultsControllerDelegate` protocol will be called:

- controllerWillChangeContent(_:)
- controller(_:didChangeObject:atIndexPath:forChangeType:newIndexPath:)
- controllerDidChangeContent(_:)

Here is the implementation. Insert the following code in the `RestaurantTableViewController` class:

```
func controllerWillChangeContent(controller: NSFetchedResultsController) {
    tableView.beginUpdates()
}

func controller(controller: NSFetchedResultsController, didChangeObject
anObject: AnyObject, atIndexPath indexPath: NSIndexPath?, forChangeType type:
NSFetchedResultsChangeType, newIndexPath: NSIndexPath?) {

    switch type {
    case .Insert:
        if let _newIndexPath = newIndexPath {
            tableView.insertRowsAtIndexPaths([_newIndexPath], withRowAnimation:
.Fade)
        }
    case .Delete:
        if let _indexPath = indexPath {
            tableView.deleteRowsAtIndexPaths([_indexPath], withRowAnimation:
.Fade)
        }
    case .Update:
        if let _indexPath = indexPath {
            tableView.reloadRowsAtIndexPaths([_indexPath], withRowAnimation:
.Fade)
        }

    default:
        tableView.reloadData()
    }

    restaurants = controller.fetchedObjects as! [Restaurant]
}
```

```
func controllerDidChangeContent(controller: NSFetchedResultsController) {
    tableView.endUpdates()
}
```

The first method is called when the fetched results controller is about to start processing the content change. We simply tell the table view, "Hey, we're going to update the table. Get ready for it."

When there is any content change in the managed context (e.g. a new restaurant is saved), the second method is called. Here we determine the type of operation and proceed with the corresponding operation. For instance, we insert new rows when the type is set to `.Insert`. Because the objects in the fetched results controller are changed, we sync them with the `restaurants` array at the end of the method.

After the fetched results controller completes the change, it calls the `controllerDidChangeContent` method. We just need to tell the table view that we've completed the update and it will animate the change correspondingly.

That's it. Now run the app again and create a few restaurants. The app should respond to the change instantaneously.

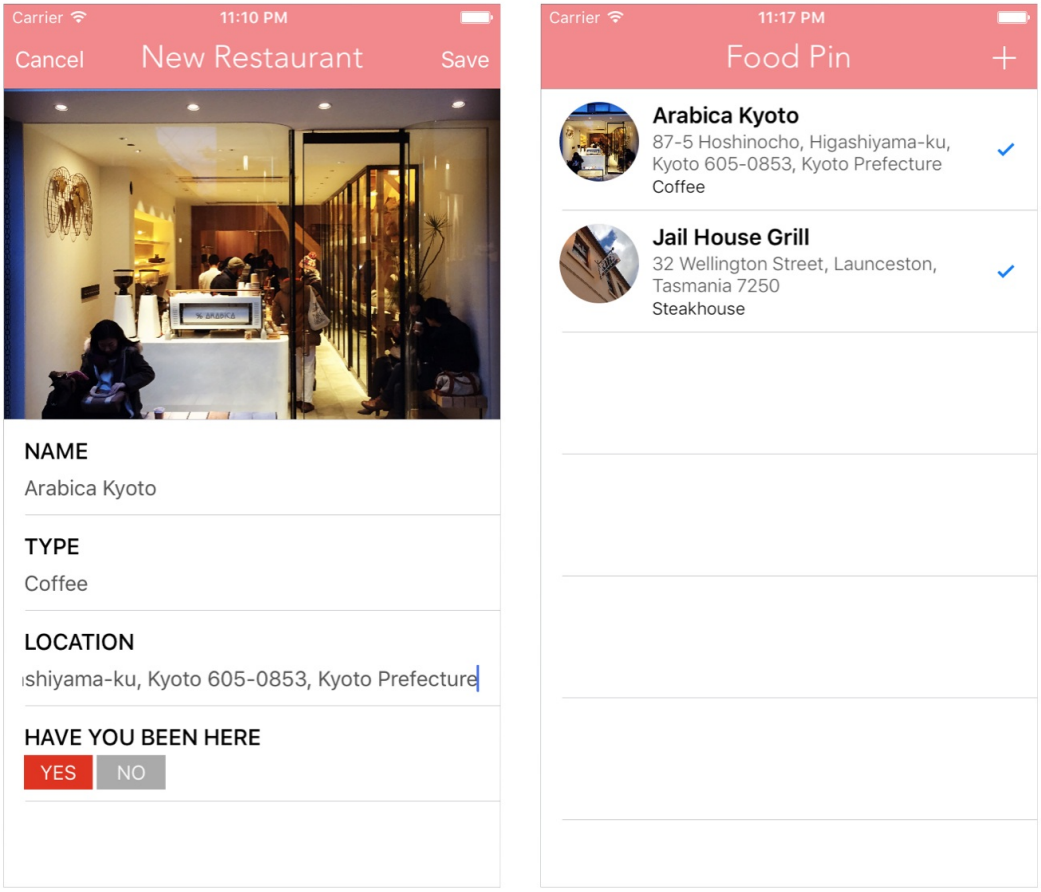

Figure 19-8. Saving a new restaurant

Deleting Data Using Core Data

Similar to data insertion, you just need to call a method named `deleteObject` with the managed object to delete. Again, you call the `save` method to apply the changes. Here is a sample code snippet:

```
managedObjectContext.deleteObject(restaurantToDelete)
```

To remove the selected restaurant from database, you simply update the `deleteAction` variable in the `tableView(_:editActionsForRowAtIndexPath:_)` method like this:

```
let deleteAction = UITableViewRowAction(style:
UITableViewRowActionStyle.Default, title: "Delete",handler: { (action,
indexPath) -> Void in

    // Delete the row from the database
```

```
    if let managedObjectContext = (UIApplication.sharedApplication().delegate
as? AppDelegate)?.managedObjectContext {

        let restaurantToDelete =
self.fetchResultController.objectAtIndexPath(indexPath) as! Restaurant
        managedObjectContext.deleteObject(restaurantToDelete)

        do {
            try managedObjectContext.save()
        } catch {
            print(error)
        }
    }
})
```

Now compile and run the app again. At this point, if you delete a record it should disappear completely from the database.

Updating a Managed Object

Updating a managed object works the same like inserting a managed object except that you do not need to call the `insertNewObjectForEntityForName` method. If you already get hold of the managed object, you can simply update the properties and call the `save` method to write the data into the database.

For example, to save the rating of a restaurant, you can update the `close` method of the `RestaurantDetailViewController` class like this:

```
@IBAction func close(segue:UIStoryboardSegue) {
    if let reviewViewController = segue.sourceViewController as?
ReviewViewController {
        if let rating = reviewViewController.rating {
            restaurant.rating = rating
            ratingButton.setImage(UIImage(named: rating), forState:
UIControlState.Normal)

            if let managedObjectContext =
(UIApplication.sharedApplication().delegate as?
AppDelegate)?.managedObjectContext {

                do {
                    try managedObjectContext.save()
                } catch {
                    print(error)
```

```
                }
            }
        }
    }
}
```

Unlike the above scenario, if you do not get hold of the managed object, you will have to create a `NSFetchRequest` object and look it up using the `executeFetchRequest` method.

Viewing the Raw SQL Statement

One of the benefits of Core Data is that developers from learning SQL. You can interact and save data into the database without learning SQL or writing a line of SQL. However, for those with a database background, you may want to study the exact SQLs executed behind the scene; this can be very useful for debugging.

Xcode allows developers to enable SQL output for debugging purposes. To enable this option, click FoodPin (right next to the Stop button) and select *Edit Scheme*.

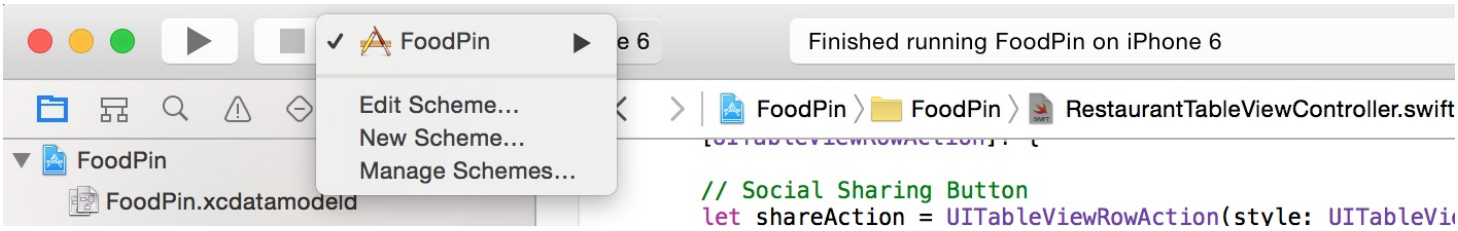

Figure 19-9. Edit scheme in Xcode project

Select the *Arguments* tab. Under the *Argument Passed on Launch* section, click the `+` button and add the `-com.apple.CoreData.SQLDebug 1` parameter.

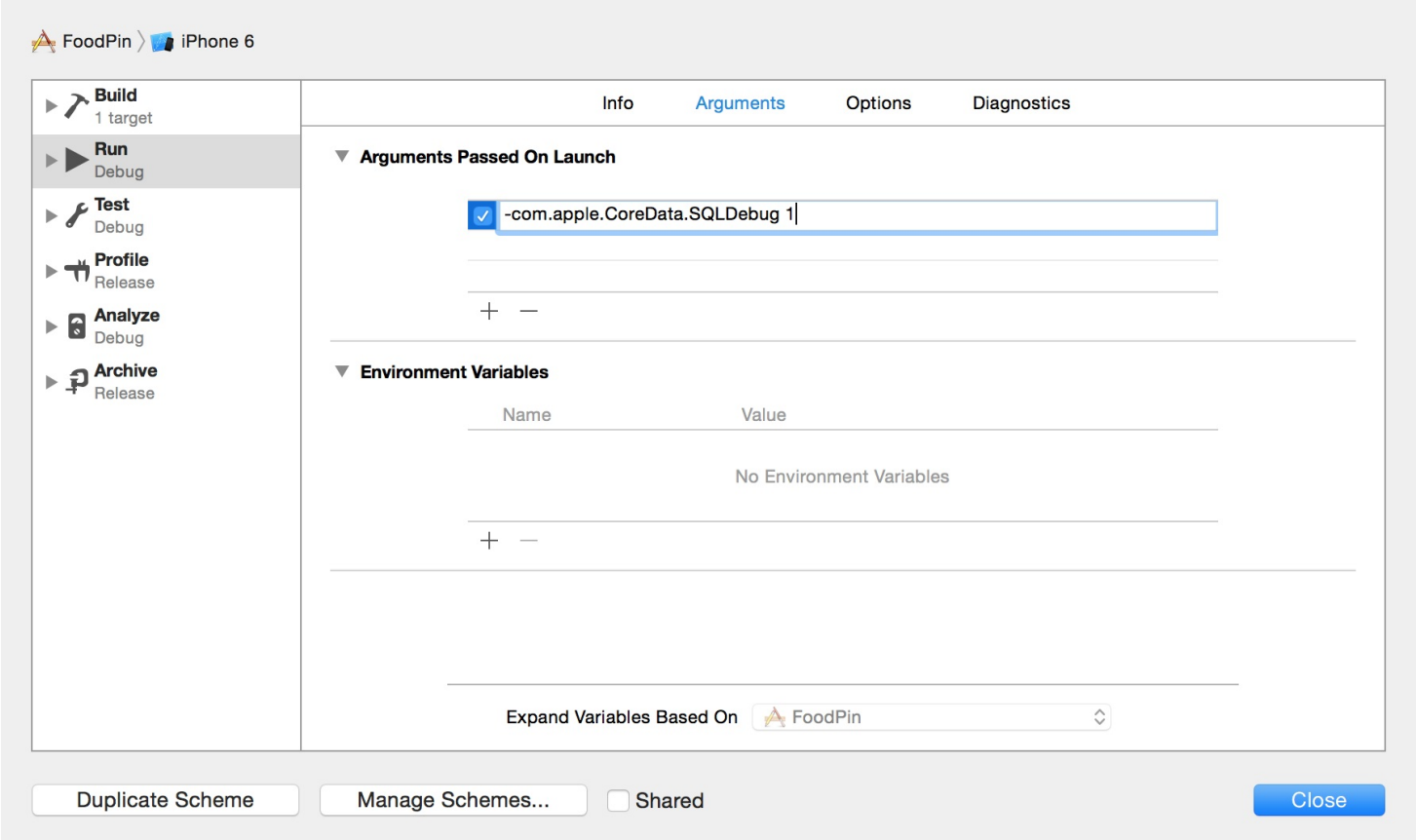

Figure 19-10. Add SQL Debug Parameter

Click `OK` to confirm. Now run your app again and you should see the raw SQL statement (e.g. SELECT and DELETE) displayed in the console window. Here is a sample output:

```
2015-09-02 00:15:30.990 FoodPin[1320:120479] CoreData: sql: SELECT 0, t0.Z_PK,
t0.Z_OPT, t0.ZIMAGE, t0.ZISVISITED, t0.ZLOCATION, t0.ZNAME, t0.ZPHONENUMBER,
t0.ZRATING, t0.ZTYPE FROM ZRESTAURANT t0 WHERE  t0.Z_PK IN  (?)
2015-09-02 00:15:31.437 FoodPin[1320:120479] CoreData: annotation: sql
connection fetch time: 0.4425s
2015-09-02 00:15:31.437 FoodPin[1320:120479] CoreData: annotation: total fetch
execution time: 0.4468s for 1 rows.
2015-09-02 00:15:31.439 FoodPin[1320:120479] CoreData: sql: BEGIN EXCLUSIVE
2015-09-02 00:15:31.440 FoodPin[1320:120479] CoreData: sql: DELETE FROM
ZRESTAURANT WHERE Z_PK = ? AND Z_OPT = ?
2015-09-02 00:15:31.444 FoodPin[1320:120479] CoreData: sql: COMMIT
```

If you're new to SQL and database, you may not understand the debugging information. SQL is a language for interacting with relational database. The SELECT statement is used for querying and selecting data from database. When you need to remove a record from database, you use

the DELETE statement.

Still confused? Don't worry, just ignore the SQL statements. This is why iOS SDK comes with the Core Data framework to shield you from the internals.

Your Exercise

I intentionally left out the phone number field for your implementation. Currently, there is no way for users to input a phone number. Your exercise is to add the *Phone Number* field in the New Restaurant screen, and save the information through Core Data.

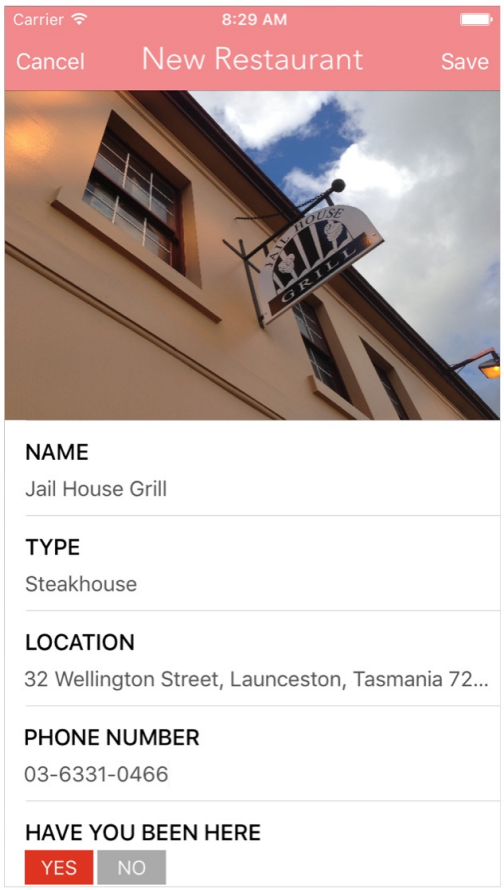

Figure 19-11. Adding a Phone Number field in the New Restaurant screen

Summary

Congratulate yourself on making an app using Core Data. At this point, you should know how

to retrieve and manage data in a persistent data store. Core Data is a powerful framework for working with persistent data especially for those who do not have any database knowledge. Our Core Data chapter ends here. Now that you understand the basics of Core Data, don't stop learning and exploring. You can check out Apple's official reference to learn more. (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html).

For reference, you can download the complete Xcode from https://www.dropbox.com/s/s02eknhxv4ogj1z/FoodPinCoreDataFinal.zip?dl=0.

Are you ready to further improve the app? I hope you're still with me. Let's move on and take a look how to add a search bar.