CHAPTER 31

HOW TO PRELOAD A SQLITE DATABASE USING CORE DATA

WORKING WITH EXISTING DATA IN CORE DATA

When working with Core Data, you may have asked these two questions:

1. How can you preload existing data into the SQLite database?

2. How can you use an existing SQLite database in my Xcode project?

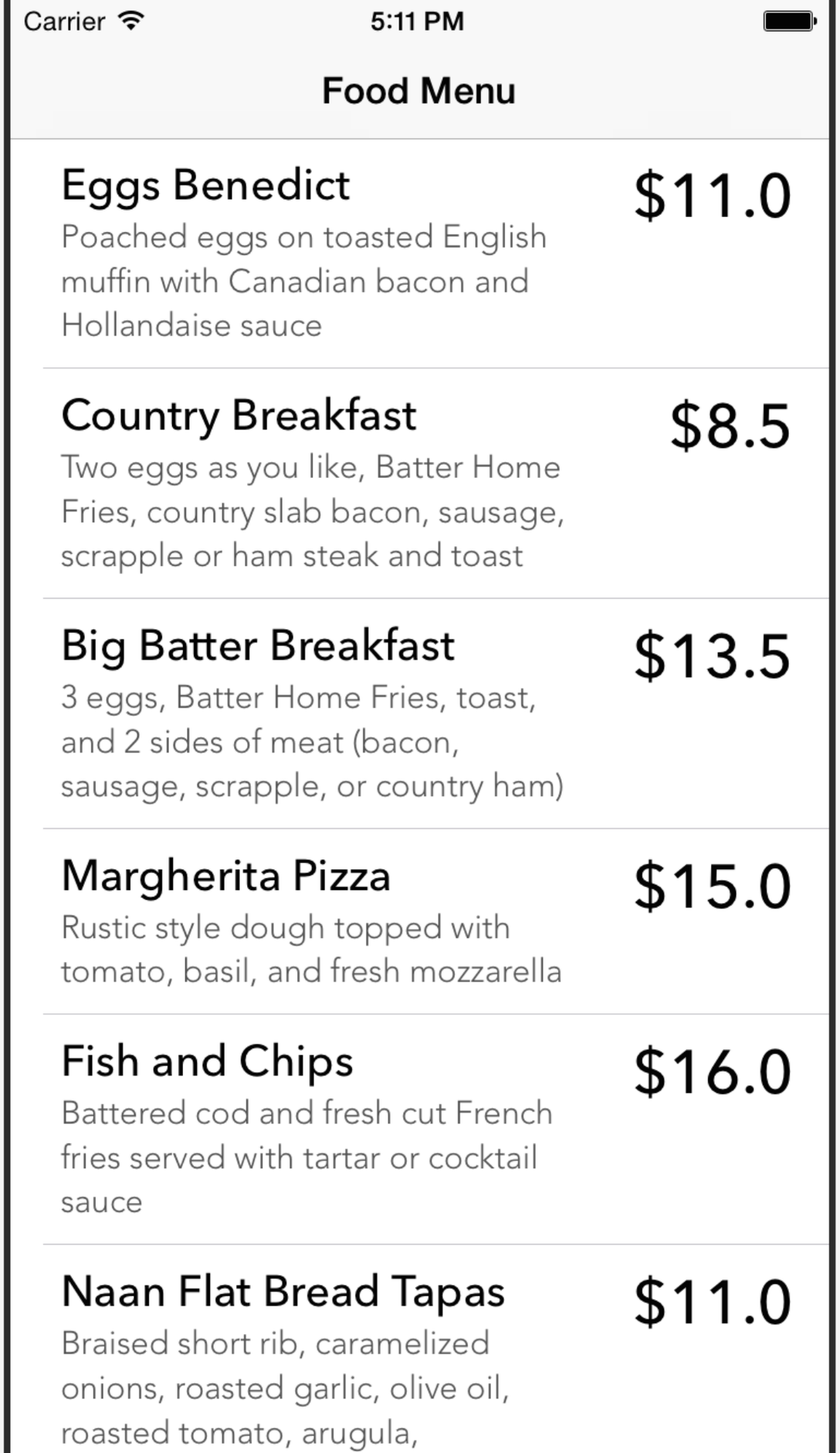
I recently met a friend who is now working on a dictionary app for a particular industry. He got the same questions. He knows how to save data into the database and retrieve them back from the Core Data store. The real question is: how could he preload the existing dictionary data into the database?

I believe some of you may have the same question. This is why I devote a full chapter to talk about data preloading in Core Data. I will answer the above questions and show you how to preload your app with existing data.

So how can you preload existing data into the built-in SQLite database of your app? In general you bundle a data file (in CSV or JSON format or whatever format you like). When the user launches the app for the very first time, it preloads the data from the data file and puts them into the database. At the time when the app is fully launched, it will be able to use the database, which has been pre-filled with data. The data file can be either bundled in the app or hosted on a cloud server. By storing the file in the cloud or other external sources, this would allow you to update the data easily, without rebuilding the app. I will walk you through both approaches by building a simple demo app.

Once you understand how data preloading works, I will show you how to use an existing SQLite database (again pre-filled with data) in your app.

Note that I assume you have a basic understanding of Core Data. You should know how to insert and retrieve data through Core Data. If you have no ideas about these operations, you can refer to Chapter 16 of the book, Beginning iOS 8 Programming with Swift.



A SIMPLE DEMO APP

To keep your focus on learning data preloading, I have created the project template for you. Firstly, download the project from https://www.dropbox.com/s/l3m5gky5qu959jt/CoreDataPreloadDemoStart.zip?dl=0 and have a trial run.

It's a very simple app showing a list of food. By default, the starter project comes with an empty database. When you compile and launch the app, your app will end up a blank table view. What we are going to do is to preload the database with existing data.

I have already built the data model and provided the implementation of the table view. You can look into the *MenuItemTableViewController* class and *CoreDataDemo.xcdatamodeld* for details. The data model is pretty simple. I have defined a *MenuItem* entity, which includes three attributes: name, detail, and price.

Once you're able to preload the database with the food menu items, the app will display them accordingly, with the resulting user interface similar to the screenshot shown on the left.

THE CSV FILE

In this demo I use a CSV file to store the existing data. CSV files are often used to store tabular data and can be easily created using text editor, Numbers or MS Excel. They are sometimes known as comma delimited files. Each record is one line and fields are separated with commas. In the project template, you should find the “menudata.csv” file. It contains all the food items for the demo app in CSV format. Here is a part of the file:

```
Eggs Benedict,"Poached eggs on toasted English muffin with Canadian bacon and Hollandaise sauce",11.0
Country Breakfast,"Two eggs as you like, Batter Home Fries, country slab bacon, sausage, scrapple or ham steak and toast", 8.5
Big Batter Breakfast,"3 eggs, Batter Home Fries, toast, and 2 sides of meat (bacon, sausage, scrapple, or country ham)",13.5
Margherita Pizza,"Rustic style dough topped with tomato, basil, and fresh mozzarella",15.0
Fish and Chips,Battered cod and fresh cut French fries served with tartar or cocktail sauce,16.0
```

The first field represents the name of the food menu item. The next field is the detail of the food, while the last field is the price. Each food item is one line, separated with a new line separator.

PARSING CSV FILES

It's not required to use CSV files to store your data. JSON and XML are two common formats for data interchange and flat file storage. As compared to CSV format, they are more readable and suitable for storing structured data. Anyway, CSV has been around for a long time and is supported by most spreadsheet applications. At some point of time, you will have to deal with this type of file. So I pick it as an example. Let's see how we can parse the data from CSV.

The AppDelegate object is normally used to perform tasks during application startup (and shutdown). To preload data during the app launch, we will add a few methods in the AppDelegate class. First, insert the following method for parsing the CSV file:

```
func parseCSV (contentsOfURL: NSURL, encoding: NSStringEncoding, error: NSErrorPointer) -> [(name:String, detail:String, price: String)]? {
    // Load the CSV file and parse it
    let delimiter = ","
    var items:[(name:String, detail:String, price: String)]?

    if let content = String(contentsOfURL: contentsOfURL, encoding: encoding, error: error) {
        items = []
    let lines:[String] = content.componentsSeparatedByCharactersInSet(NSCharacterSet.newlineCharacterSet()) as [String]

    for line in lines {
        var values:[String] = []
        if line != "" {
            // For a line with double quotes
            // we use NSScanner to perform the parsing
            if line.rangeOfString("\"") != nil {
                var textToScan:String = line
                var value:NSString?
                var textScanner:NSScanner = NSScanner(string: textToScan)
                while textScanner.string != "" {

                    if (textScanner.string as NSString).substringToIndex(1) == "\"" {
                        textScanner.scanLocation += 1
                        textScanner.scanUpToString("\"", intoString: &value)
```

```
            textScanner.scanLocation += 1
        } else {
            textScanner.scanUpToString(delimiter, intoString: &value)
        }

        // Store the value into the values array
        values.append(value as! String)

        // Retrieve the unscanned remainder of the string
        if textScanner.scanLocation < count(textScanner.string) {
            textToScan = (textScanner.string as NSString).substringFromIndex(textScanner.scanLocation + 1)
        } else {
            textToScan = " "
        }
        textScanner = NSScanner(string: textToScan)
    }

            // For a line without double quotes, we can simply separate the string
            // by using the delimiter (e.g. comma)
            } else  {
                values = line.componentsSeparatedByString(delimiter)
            }

            // Put the values into the tuple and add it to the items array
            let item = (name: values[0], detail: values[1], price: values[2])
            items?.append(item)
        }
    }
  }

  return items
}
```

The method takes in three parameters: the file's URL, encoding and an error pointer. It first loads the file content into memory, reads the lines into an array and then performs the parsing line by line. At the end of the method, it returns an array of food menu items in the form of tuples.

A simple CSV file only uses a comma to separate values. Parsing such kind of CSV files shouldn't be difficult. You can call the *componentsSeparatedByString* method (highlighted in yellow in the code snippet) to split a comma-delimited string. It'll then return you an array of strings that have been divided by the separator.

For some CSV files, they are more complicated. Field values containing reserved characters (e.g. comma) are surrounded by double quotes. Here is another example:

```
Country Breakfast,"Two eggs as you like, Batter Home Fries, country slab bacon, sausage, scrapple or ham steak and toast", 8.5
```

In this case, we cannot simply use the *componentsSeparatedByString* method to separate the field values. Instead, we use NSScanner to go through each character of the string and retrieve the field values. If the field value begins with a double quote, we scan through the string until we find the next double quote character by calling the *scanUpToString* method. The method is smart enough to extract the value surrounded by the double quotes. Once a field value is retrieved, we then repeat the same procedure for the remainder of the string.

After all the field values are retrieved, we save them into a tuple and then put it into the "items" array.

PRELOADING THE DATA AND SAVING IT INTO DATABASE

Now that you've created the method for CSV parsing, we now move onto the implementation of data preloading. The preloading will work like this:

1. First, we will remove all the existing data from the database. This operation is optional if you can ensure the database is empty.
2. Next, we will call up the parseCSV method to parse menudata.csv.
3. Once the parsing completes, we insert the food menu items into the database.

Add the following code snippets into the AppDelegate.swift file:

```
func preloadData () {

    // Retrieve data from the source file
    if let contentsOfURL = NSBundle.mainBundle().URLForResource("menudata", withExtension: "csv") {

        // Remove all the menu items before preloading
        removeData()

        var error:NSError?
        if let items = parseCSV(contentsOfURL, encoding: NSUTF8StringEncoding, error: &error) {
            // Preload the menu items
            if let managedObjectContext = self.managedObjectContext {
                for item in items {
                    let menuItem = NSEntityDescription.insertNewObjectForEntityForName("MenuItem", inManagedObjectContext:
managedObjectContext) as! MenuItem
                    menuItem.name = item.name
                    menuItem.detail = item.detail
                    menuItem.price = (item.price as NSString).doubleValue

                    if managedObjectContext.save(&error) != true {
                        println("insert error: \(error!.localizedDescription)")
                    }
```

```
            }
        }
    }
  }
}

func removeData () {
    // Remove the existing items
    if let managedObjectContext = self.managedObjectContext {
        let fetchRequest = NSFetchRequest(entityName: "MenuItem")
        var e: NSError?
        let menuItems = managedObjectContext.executeFetchRequest(fetchRequest, error: &e) as! [MenuItem]

        if e != nil {
            println("Failed to retrieve record: \(e!.localizedDescription)")

        } else {

            for menuItem in menuItems {
                managedObjectContext.deleteObject(menuItem)
            }
        }
    }
}
```

The *removeData* method is used to remove any existing menu items from the database. I want to ensure the database is empty before populating the data extracted from the menudata.csv file. The implementation of the method is very straightforward if you have a basic understanding of Core Data. We first execute a query to retrieve all the menu items from the database and call the *deleteObject* method to delete the item one by one.

Okay, now let's talk about the *preloadData* method.

In the method we first retrieve the file URL of the menudata.csv file using this line of code:

```
NSBundle.mainBundle().URLForResource("menudata", withExtension: "csv")
```

After calling the *removeData* method, we execute the parseCSV method to parse the menudata.csv file. With the returned items, we insert them one by one by calling the *NSEntityDescription.insertNewObjectForEntityForName* method.

Lastly, execute the *preloadData()* method in the *didFinishLaunchingWithOptions* method:

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {

    preloadData()

    return true
}
```

Now you're ready to test your app. Hit the Run button to launch the app. If you've followed the implementation correctly, the app should be preloaded with the food items.

But there is an issue with the current implementation. Every time you launch the app, it preloads the data from the CSV file. Apparently, you only want to perform the preloading once. Change the *application:didFinishLaunchingWithOptions:* method to the following:

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {

    let defaults = NSUserDefaults.standardUserDefaults()
    let isPreloaded = defaults.boolForKey("isPreloaded")
    if !isPreloaded {
        preloadData()
        defaults.setBool(true, forKey: "isPreloaded")
    }

    return true
}
```

To indicate that the app has preloaded the data, we save a setting to the defaults system using a specific key (i.e. isPreloaded). Every time when the app is launched, we will first check if the value of the "isPreloaded" key. If it's set to true, we will skip the data preloading operation.

USING EXTERNAL DATA FILES

So far the CSV file is bundled in the app. If your data is static, it is completely fine. But what if you're going to change the data frequently? In this case, whenever there is a new update for the data file, you will have to rebuild the app and redeploy it to the app store.

There is a better way to handle this.

Instead of embedding the data file in the app, you put it in an external source. For example, you can store it on a cloud server. Every time when a user opens the app, it goes up to the server and download the data file. Then the app parses the file and loads the data into the database as usual.

I have uploaded the sample data file to the Amazon cloud server. You can access it through the URL below:

https://s3.amazonaws.com/swiftbook/menudata.csv

This is just for demo purpose. If you have your own server, feel free to upload the file to the server and use your own URL. To load the data file from the remote server, all you need to do is make a little tweak to the code. First, update the *preloadData* method to the following:

```
func preloadData () {

    // Remove all the menu items before preloading
    removeData()

    var error:NSError?
    let remoteURL = NSURL(string: "https://s3.amazonaws.com/swiftbook/menudata.csv")!
    if let items = parseCSV(remoteURL, encoding: NSUTF8StringEncoding, error: &error) {
        // Preload the menu items
        if let managedObjectContext = self.managedObjectContext {
            for item in items {
                let menuItem = NSEntityDescription.insertNewObjectForEntityForName("MenuItem", inManagedObjectContext: managedObjectContext)
as! MenuItem
                menuItem.name = item.name
```


```
            menuItem.detail = item.detail
            menuItem.price = (item.price as NSString).doubleValue

            if managedObjectContext.save(&error) != true {
                println("insert error: \(error!.localizedDescription)")
            }
        }
    }
  }
}
```

The code is very similar to the original one. Instead loading the data file from the bundle, we specify the remote URL and pass it to the *parseCSV* method. That's it. The *parseCSV* method will handle the file download and perform the data parsing accordingly.

Before running the app, you have to update the *application:didFinishLaunchingWithOptions:* method so that the app will load the data every time it runs:

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[NSObject: AnyObject]?) -> Bool {

    preloadData()

    return true
}
```

You're ready to go. Hit the Run button and test the app again. The menu items should be different from those shown previously.

For your reference, you can download the complete Xcode project from https://www.dropbox.com/s/0w12gmg1ldxuui5/CoreDataPreloadDemo.zip?dl=0.

USING AN EXISTING DATABASE IN YOUR PROJECT

Now that you should know how to populate a database with external data, you may wonder if you can use an existing SQLite database directly. In some situations, you probably do not want to preload the data during app launch. For example, you need to preload hundreds of thousands of records. This will take some time to load the data and results a poor user experience. Apparently, you want to pre-filled the database beforehand and bundle it directly in the app.

Suppose you've already pre-filled an existing database with data, how can you bundle it in your app?

Before I show you the procedures, please download the starter project again from https://www.dropbox.com/s/l3m5gky5qu959jt/CoreDataPreloadDemoStart.zip?dl=0. As a demo, we will copy the existing database created in the previous section to this starter project.

Now open up the Xcode project that you have worked on earlier. If you've followed me along, your database should be pre-filled with data. We will now copy it to the starter project that you have just downloaded.

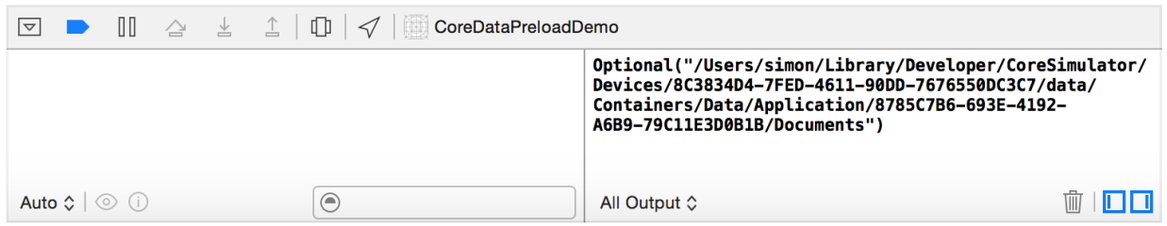
But where is the SQLite database?

The database is not bundled in the Xcode project but automatically created when you run the app in the simulator. To locate the database, you will need to add a line of code to reveal the file path. Update the *application:didFinishLaunchingWithOptions:* method to the following:

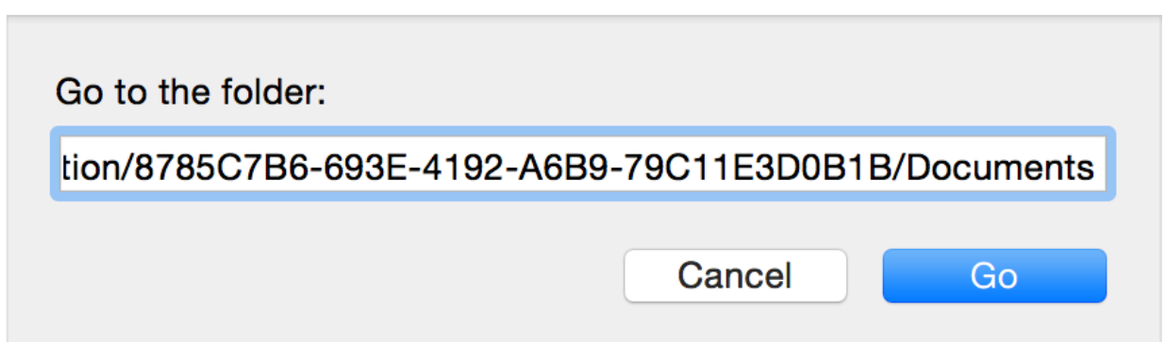
```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
-> Bool {

    println(applicationDocumentsDirectory.path)
    preloadData()

    return true
}
```

The SQLite database is generated under the application's document directory. To find the file path, we simply print out the value of *applicationDocumentsDirectory.path* variable.

Now run the app again. You should see an output in the console window showing the full path of the document directory.



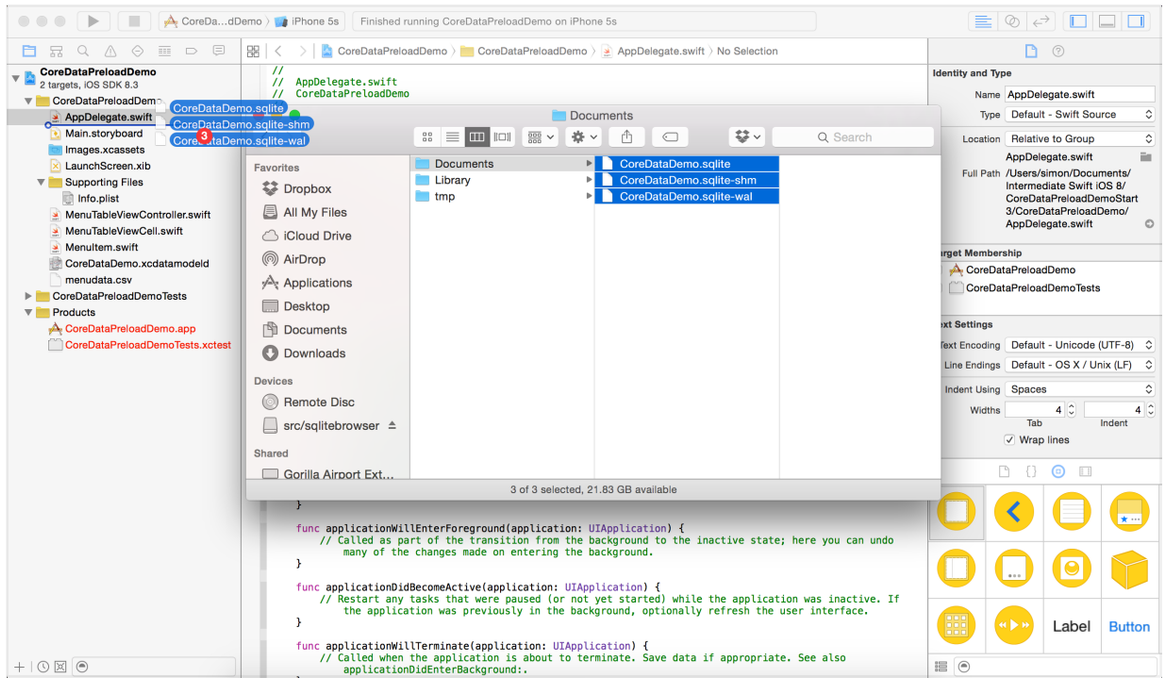
Copy the file path and go to Finder. In the menu select Go > Go to Folder... and then paste the path in the pop-up. Click "Go" to confirm.



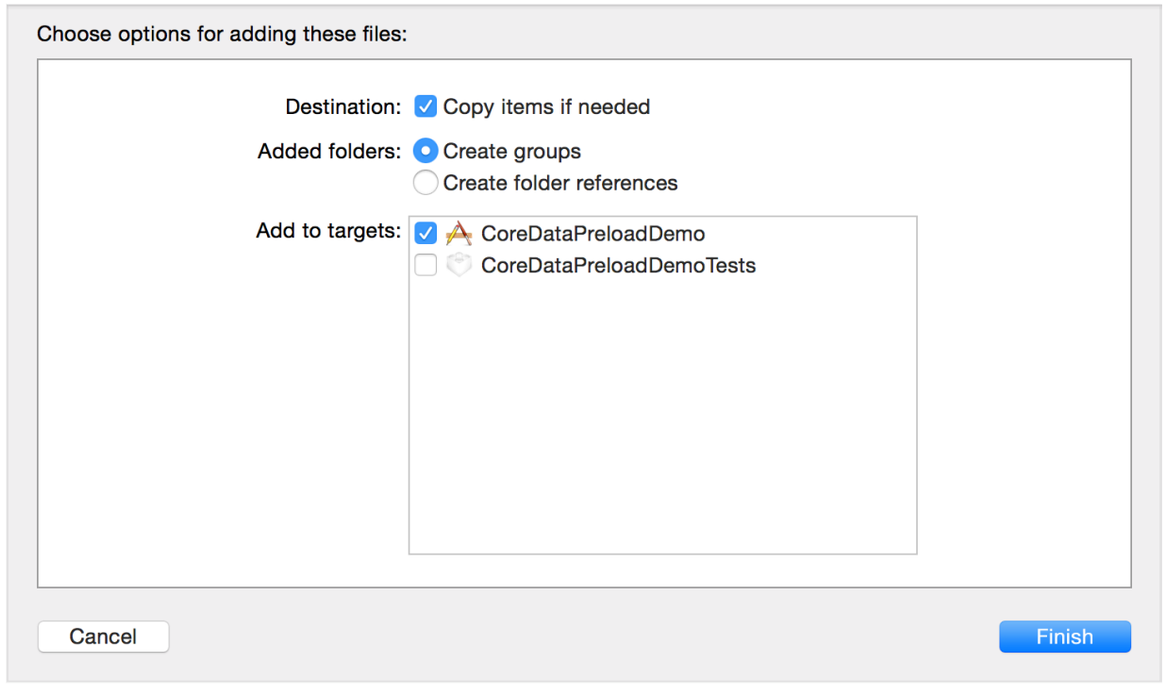
Once you open the document folder in Finder, you will find three files: CoreDataDemo.sqlite, CoreDataDemo.sqlite-wal and CoreDataDemo.sqlite-shm.

Starting from iOS 7, the default journaling mode for Core Data SQLite stores is set to Write-Ahead Logging (WAL). With the WAL mode, Core Data keeps the main .sqlite file untouched and appends transactions to a .sqlite-wal file in the same folder. When running WAL mode, SQLite will also create a shared memory file with .sqlite-shm extension. In order to backup the database or use it to in other projects, you will need copy these three files. If you just copy the CoreDataDemo.sqlite file, you will probably end up with an empty database.

Now, drag these three files to the starter project in Xcode.



When prompted, please ensure the “Copy item if needed” option is checked and the “CoreDataPreloadDemo” option of “Add to Targets” is selected. Then click “Finish” to confirm.



Now that you've bundled an existing database in your Xcode project. When you build the app, this database will be embedded in the app. But you will have to tweak the code a bit before the app is able to use the database.

By default, the app will create an empty SQLite store if there is no database found in the document directory. So all you need to do is copy the database files bundled in the app to that directory. In the AppDelegate class update the declaration of the persistentStoreCoordinator variable like this:

```
lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator? = {
    // The persistent store coordinator for the application. This implementation creates and return 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
    var coordinator: NSPersistentStoreCoordinator? = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataDemo.sqlite")

    if !NSFileManager.defaultManager().fileExistsAtPath(url.path!) {
        let sourceSqliteURLs = [NSBundle.mainBundle().URLForResource("CoreDataDemo", withExtension: "sqlite")!, NSBundle.mainBundle().URLForResource("CoreDataDemo", withExtension: "sqlite-wal")!, NSBundle.mainBundle().URLForResource("CoreDataDemo", withExtension: "sqlite-shm")!]

        let destSqliteURLs = [self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataDemo.sqlite"),
            self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataDemo.sqlite-wal"),
            self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataDemo.sqlite-shm")]

        var error:NSError? = nil
        for var index = 0; index < sourceSqliteURLs.count; index++ {
            NSFileManager.defaultManager().copyItemAtURL(sourceSqliteURLs[index], toURL: destSqliteURLs[index], error: &error)
        }
    }

    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
```

```
        // 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
        error = 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 \(error), \(error!.userInfo)")
        abort()
    }

    return coordinator
    }()
```

The changes are highlighted in yellow. We first verify if the database exists in the document folder. If not, we copy the SQLite files from the bundle folder to the document folder by calling the *copyItemAtURL* method of NSFileManager.

That's it! Before you hit the Run button to test the app, you better delete the CoreDataPreloadDemo app from the simulator or simply reset it (select iOS Simulator > Reset Content and Settings). This is to remove any existing SQLite databases from the simulator.

Okay, now you're good to go. When the app is launched, it should be able to use the database bundled in the Xcode project.

For reference, you can download the final Xcode project from https://www.dropbox.com/s/ox0878t9k0grsek/CoreDataExistingDB.zip?dl=0.