Extracted from:

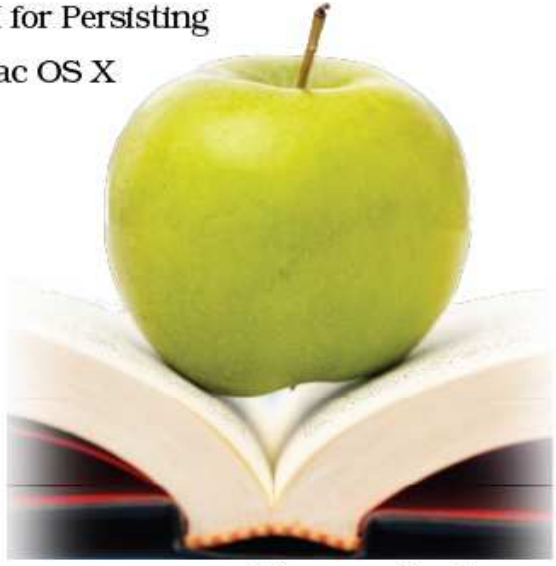
Core Data

Apple's API for Persisting Data on Mac OS X

The Pragmatic Programmers

Core Data

Apple's API for Persisting Data on Mac OS X



Marcus S. Zarra

Edited by Daniel H Steinberg

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-32-8

ISBN-13: 978-1-934356-32-6

Printed on acid-free paper.

B6.0 printing, July 29, 2009

Version: 2009-8-25

Chapter 10

Core Data and iPhone

Core Data is now available on the iPhone. Introduced as part of the iPhone 3.0 SDK, the API is nearly identical to the desktop version. There are, however, some very important differences that we will review in this chapter.

This chapter assumes that you have at least a basic understanding of how code is written for the Cocoa Touch devices and are comfortable with the UIViewController design. If you are not, then I highly recommend reading *iPhone SDK Development* [**?**]before proceeding with this chapter.

10.1 Similarities and Differences

The Core Data API is nearly identical on both the desktop and Cocoa Touch devices. Of course, "nearly identical" and "identical" are not the same thing. We need to be conscious of a few very important differences between the desktop and Cocoa Touch before designing an application to run on Cocoa Touch.

Creating a New Core Data Cocoa Touch Project

When starting a new Cocoa Touch project, it is possible to add Core Data to many of the existing templates. To do this, select the template that you want to start with, and then select the "Use Core Data for storage" checkbox before progressing in the creation of the template.

To demonstrate using Core Data on the iPhone, we will be using the Navigation-based Application template with the "Use Core Data for storage" box selected, as shown in Figure 10.1, on the next page. We will be starting this project in Section 10.4, *Recipes for the iPhone*, on page 195.

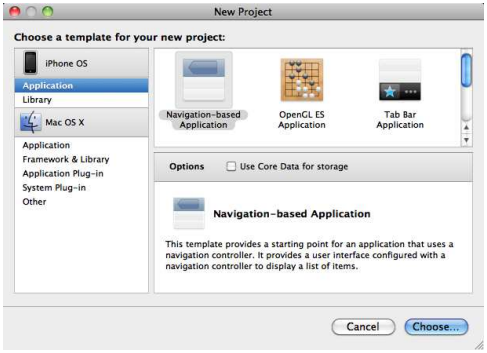


Figure 10.1: iPhone OS New Project dialog box

Upgrading an Existing Application to Core Data

Many of us have been developing iPhone applications since the SDK's original release, so it is quite likely that you have an existing application that you want to integrate with Core Data. Fortunately, it does not take very much code to build up the Core Data stack in an existing project.

Adding a Data Model to the Project

Just like on the desktop, Core Data on Cocoa Touch requires a data model to define the structure of the data entities. Therefore, the first step is to add a data model to the project by selecting File > New File. Within the dialog box that appears, if we select the Resources section, we can then create a new data model, as shown in Figure 10.2, on the next page.

In addition to creating a new data model, we can also use an existing data model from an existing application. Later in this chapter, in Section 10.4, *Recipes for the iPhone*, on page 195, we use the data model from our desktop recipes application in our new iPhone application. The data models are compatible between the desktop and Cocoa Touch, which allows us to share not only models but the underling persistent stores as well.

Adding the Core Data Code

Once we have a data model to work with, we next need to add the code to load the Core Data stack. The exact placement of this code depends on the design of your application. Since there are no document-style

Figure 10.2: Adding a data model to the project

applications on the iPhone, it is most common to have a single data model and a single persistent store per Cocoa Touch application. Although I still like to put the Core Data code in the application delegate, it might make sense to put it somewhere else. No matter where the code is placed, it is very similar to the desktop. First we need to initialize the data model.

Download RecipeCT/Classes/AppDelegate.m

```
- (NSManagedObjectModel*)managedObjectModel
{
  if (managedObjectModel) return managedObjectModel;

  NSString *path = [[NSBundle mainBundle] pathForResource:@"DataModel"
                                                   ofType:@"momd"];
  if (!path) {
    path = [[NSBundle mainBundle] pathForResource:@"DataModel"
                                           ofType:@"mom"];
  }
  NSAssert(path != nil, @"Unable to find DataModel in main bundle");
  NSURL *url = [NSURL fileURLWithPath:path];
  managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:url];
  return managedObjectModel;
}
```

This could should look quite familiar because it is identical to the way you would build the NSManagedObjectModel on the desktop. We get the path for the .mom file (or the .momd if there are multiple versions of the data model) and use it to initialize the NSManagedObjectModel.

Download RecipeCT/Classes/AppDelegate.m

```
- (NSString*)documentsFolder
{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                       NSUserDomainMask, YES);
  NSString *filePath = [paths objectAtIndex:0];
  return filePath;
}
```

Before we construct the NSPersistentStoreCoordinator, we need to decide where to store the persistent store file. On the desktop in an application with a single persistent store, we would save the file to the Application Support folder. However, on Cocoa Touch devices, there is no such location. Instead, each application has its own sandboxed Documents directory designed for the storage of files. This is where we will write our persistent store. Using code similar to what we used on the desktop to find the Application Support folder, we will find the Documents folder specific to our application.

Download RecipeCT/Classes/AppDelegate.m

```
- (NSPersistentStoreCoordinator*)persistentStoreCoordinator;
{
  if (persistentStoreCoordinator) return persistentStoreCoordinator;

  NSFileManager *fileManager = [NSFileManager defaultManager];
  NSString *docFolder = [self documentsFolder];
  if (![fileManager fileExistsAtPath:docFolder]) {
    [fileManager createDirectoryAtPath:docFolder attributes:nil];
  }

  NSString *filePath = nil;
  filePath = [docFolder stringByAppendingPathComponent:@"recipes.sqlite"];
  if (![fileManager fileExistsAtPath:filePath]) {
    NSString *defaultDB = [[NSBundle mainBundle] pathForResource:@"recipes"
                                                          ofType:@"sqlite"];
    NSError *error = nil;
    if (![[NSFileManager defaultManager] copyItemAtPath:defaultDB
                                                 toPath:filePath
                                                  error:&error]) {
      NSLog(@"%@:%s Error copying file %@", [self class], _cmd, error);
    }
  }

  NSURL *url = [NSURL fileURLWithPath:filePath];
  NSManagedObjectModel *mom = [self managedObjectModel];
  persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                                initWithManagedObjectModel:mom];

  NSError *error = nil;
  if ([persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
```

```
                                                    configuration:nil
                                                              URL:url
                                                          options:nil
                                                            error:&error]) {
    return persistentStoreCoordinator;
  }

  [persistentStoreCoordinator release], persistentStoreCoordinator = nil;
  NSDictionary *ui = [error userInfo];
  if (![ui valueForKey:NSDetailedErrorsKey]) {
    NSLog(@"%@:%s Error adding store %@", [self class], _cmd,
          [error localizedDescription]);
  } else {
    for (NSError *suberror in [ui valueForKey:NSDetailedErrorsKey]) {
      NSLog(@"%@:%s Error: %@", [self class], _cmd,
            [suberror localizedDescription]);
    }
  }
  NSAssert(NO, @"Failed to initialize the persistent store");
  return nil;
}
```

Once we have the NSManagedObjectModel constructed, the next step is to build the persistent store coordinator. Again, this code is nearly identical to the desktop version but with a few differences. First, for the moment, I have turned off the versioning check because we have only one version on the iPhone. When we release version 2 in the future, we will need to turn that back on.

The second major difference has to do with default settings. We initially check for the existence of a database file in the application's Documents directory, but if it does not exist, then we copy one from within the bundle of the application itself. This gives us a set of "defaults" or "samples" for the user who is accessing the iPhone application for the very first time. By doing this, we can present the user with an inviting list of recipes when they launch our application instead of an empty table view.

Download RecipeCT/Classes/AppDelegate.m

```
- (NSManagedObjectContext*)managedObjectContext
{
  if (managedObjectContext) return managedObjectContext;

  NSPersistentStoreCoordinator *coord = [self persistentStoreCoordinator];
  if (!coord) return nil;

  managedObjectContext = [[NSManagedObjectContext alloc] init];
  [managedObjectContext setPersistentStoreCoordinator:coord];
```

```
  return managedObjectContext;
}
```

The last method we need to implement is the -managedObjectContext method. Since we did all the hard work either in the -managedObjectModel method or in the -persistentStoreCoordinator method, this method is even simpler than its desktop cousin. We request a reference to the NSPersistentStoreCoordinator, and assuming that it is not nil, we initialize an NSManagedObjectContext, add the NSPersistentStoreCoordinator to it, and return the resulting NSManagedObjectContext. Since we will either have an existing persistent store from the last time the user ran the application or have a default store copied over, there is no need to check the Type table as we have previously. It is guaranteed either to be there or to be intentionally cleared out by the user.

Persistent Store Formats

Similar to Core Data on the desktop, several persistent formats are available on Cocoa Touch devices. However, one format is missing that I have grown to love. The XML format is not available currently on the iPhone. I suspect this is to force us to use something that is far more memory efficient such as the SQLite store. It is also possible that it was skipped because of dependencies on other APIs that are also not available at this time. Whatever the reason, the XML store is not available to us, and we should be using the SQLite store in every situation possible.

Besides the SQLite persistent store format, we also have access to the binary and in-memory formats. However, both of these formats require that the entire object hierarchy be loaded into memory, and that is something we generally cannot afford on a Cocoa Touch device. Therefore, unless there is a very solid design reason to use another store format, SQLite should be used.

10.2 Memory Management

One of the most important differences we need to keep in mind while working with Core Data on the iPhone is the management of memory. Depending on which Cocoa Touch device is targeted, we could have as little as 20MB of memory to work with. This is drastically different from the modern desktop that measures memory in gigabytes! Therefore, Core Data, to be a good citizen on this much smaller device, must handle memory differently.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Core Data's Home Page
http://pragprog.com/titles/mzcd
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/mzcd.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)