iOS5 Compliant

Store and retrieve your app data accurately and efficiently

Pro Core Data for iOS

SECOND EDITION

Michael Privat | Robert Warner

Apress®

Pro Core Data for iOS

Data Access and Persistence Engine for iPhone, iPad, and iPod touch

Second Edition

Michael Privat and
Robert Warner

Apress®

Pro Core Data for iOS: Data Access and Persistence Engine for iPhone, iPad, and iPod touch Second Edition

ISBN-13 (pbk): 978-1-4302-3656-6

ISBN-13 (electronic): 978-1-4302-3657-3

President and Publisher: Paul Manning
Lead Editor: Steve Anglin
Development Editors: Matthew Moodie and Douglas Pundick
Technical Reviewer: Robert Hamilton
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Jennifer L. Blackwell
Copy Editor: Mary Behr
Compositor: MacPS, LLC
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

To my loving wife, Kelly, and our children, Matthieu and Chloé.

—Michael Privat

To my beautiful wife, Sherry, and our wonderful children:
Tyson, Jacob, Mallory, Camie, and Leila.

—Rob Warner

Contents at a Glance

Contents

About the Authors

Michael Privat is the President and CEO of Majorspot, Inc., developer of the following iPhone and iPad apps:

- Ghostwriter Notes
- My Spending
- iBudget
- Chess Puzzle Challenge

He is also an expert developer and technical lead for Availity, LLC, based in Jacksonville, Florida. He earned his Master's degree in Computer Science from the University of Nice in Nice, France. He moved to the United States to develop software in artificial intelligence at the Massachusetts Institute of Technology. Coauthor of *Beginning OS X Lion Apps Development* (Apress, 2011), he now lives in Jacksonville, Florida, with his wife, Kelly, and their two children.

Rob Warner is a senior technical staff member for Availity, LLC, based in Jacksonville, Florida, where he works with various teams and technologies to deliver solutions in the health care sector. He coauthored *Beginning OS X Lion Apps Development* (Apress, 2011) and *The Definitive Guide to SWT and JFace* (Apress, 2004), and he blogs at `www.grailbox.com`. He earned his Bachelor's degree in English from Brigham Young University in Provo, Utah. He lives in Jacksonville, Florida, with his wife, Sherry, and their five children.

About the Technical Reviewer

Robert Hamilton is a seasoned information technology director for Blue Cross Blue Shield of Florida. He is experienced in developing applications for the iPhone and iPad; his most recent project was Ghostwriter Notes.

Before entering his leadership role at BCBSF, Robert excelled as an application developer, having envisioned and created the first claims status application used by their providers through Availity.

A native of Atlantic Beach, Florida, Robert received his B.S. in Information Systems from the University of North Florida. He supports The First Tee of Jacksonville and the Cystic Fibrosis Foundation. He is the proud father of two daughters.

Acknowledgments

There is no telling how many books never had a chance to be written because the potential authors had other family obligations to fulfill. I thank my wife, Kelly, and my children, Matthieu and Chloé, for allowing me to focus my time on this book for a few months and accomplish this challenge. Without their unconditional support and encouragement, I would not have been able to contribute to the creation of this book.

Working on this book with Rob Warner has been enlightening. I have learned a lot from him throughout this effort. His dedication to getting the job done correctly carried me when I was tired. His technical skills got me unstuck a few times when I was clueless. His gift for writing so elegantly and his patience have made my engineer jargon sound like nineteenth century prose.

I also thank the friendly and savvy Apress team who made the whole process work like a well-oiled machine. Jennifer Blackwell helped us through the entire project, guiding us through all the tasks that are required of authors. Douglas Pundick shared his editorial wisdom to keep this work readable, well organized and understandable; Steve Anglin, Matthew Moodie, Mary Behr, and the rest of the Apress folks were always around for us to lean on.

Robert Hamilton was once again a reliable watchdog to correct our technical mistakes. I'd also like to thank Brian Kohl for saving us from shaming ourselves at times with overly complicated code.

Finally, I thank the incredibly talented people of Availity who were supportive of this book from the very first day and make this company a great place to work at. Trent Gavazzi, Ben Van Maanen, Taryn Tresca, Herve Devos, and all the others offered friendship and encouragement. The last bit of thanks goes to Geoff Packwood for calling in regularly to check on the progress.

—Michael Privat

What a privilege it's been to write a second edition of *Pro Core Data for iOS*! I thank Apress for the opportunity, particularly Steve Anglin, Jennifer Blackwell, Douglas Pundick, Matthew Moodie, Mary Behr, and Robert Hamilton. It's good to get a second crack at an intriguing topic.

Thanks to everyone who read the first edition, provided feedback, posted reviews, e-mailed thanks and questions, and generally made us feel that all our efforts made a dent. We've tried to incorporate your feedback into this edition, and we welcome any praise, criticism, and questions.

I thank my wife, Sherry, and my children (Tyson, Jacob, Mallory, Camie, and Leila) for their support and encouragement. I promise to take some downtime now, at least for awhile.

Working with Michael both enlightened and humbled me. I learned so much, yet was reminded often of how much I have to learn. I thank Michael for his persistence and dedication.

Thanks also to my employer, Availity, for providing opportunities to keep my mind nimble and engaged. Naming names creates the dilemma of knowing where to stop, so I'll keep this purposely short: thanks to Trent for all the challenges, opportunities, and support. Thanks to Jon for letting me contribute to the Innovation Center. And thanks to Brian Kohl for the Code Jams! Finally, thanks to Mom, Dad, and my siblings and in-laws for asking, "How's the book coming?" and then listening to me describe all the details. Or at least pretending to.

Introduction

Interest in developing apps for Apple's iOS platform continues to rise, and more great apps appear in Apple's App Store every day. As people like you join the app-creation party, they usually discover that their apps must store data on iOS devices to be useful. Enter *Pro Core Data for iOS*, written for developers who have learned the basics of iOS development and are ready to dive deeper into topics surrounding data storage to take their apps from pretty good to great. Core Data, Apple's technology for data storage and retrieval, is both easy to approach and difficult to master. This book spans the gamut, starting you with the simple and taking you through the advanced. Read each topic, understand what it means, and incorporate it into your own Core Data apps.

Why a Second Edition?

Since the publication of the first edition of *Pro Core Data for iOS*, Apple has released Xcode 4, a major overhaul of their programming tool. Everything has moved or changed somehow, so the descriptions and tutorials from the first edition of this book, which used Xcode 3, no longer apply. All the descriptions and screenshots have been updated to the new interface.

We didn't stop at updating the book for Xcode 4, however. We broke the discussion of `NSFetchedResultsController` into its own chapter, giving it more treatment and coverage. We dug deeper into the tricky topic of migrations. We took a new approach to the section on data encryption, based on feedback from Brian Kohl. We responded to feedback we've received via reviews and e-mail. We think both new readers and people who have already read the first edition will profit from reading this edition.

What You'll Need

To follow along with this book, you need an Intel Mac running Snow Leopard or Lion, and you need Xcode 4, which is available from the Mac App Store or from `developer.apple.com` for registered Apple developers. You'll also do better if you have at least a basic understanding of Objective-C, Cocoa Touch, and iOS development.

What You'll Find

This book starts by setting a clear foundation for what Core Data is and how it works, and then it takes you step-by-step through how to get the results you need from this powerful framework. You'll learn about the components of Core Data and how they interact, how to design your data model, how to filter your results, how to tune performance, how to migrate your data across data model versions, and many other topics around and between these that will separate your apps from the crowd.

This book combines theory and code to teach its subject matter. Although you can take the book to your Barcalounger and read it from cover to cover, you'll find the book is more effective if you're in front of a computer, typing in and understanding the code it explains. We also hope that, after you read the book and work through its code, you'll keep it handy as a reference, turning to it often for answers and clarification.

How This Book Is Organized

We've tried to arrange the material so that it builds from beginning topics to advanced, at least in a general sense, as the book progresses. The topics tend to build on each other, so you'll likely benefit most by working through the book front to back, rather than skipping around. If you're looking for guidance on a specific topic—versioning and migrating data, say, or tuning performance and memory usage—skip ahead to that chapter. Most chapters focus on a single topic, indicated by that chapter's title. The final chapter covers an array of advanced topics that don't fit neatly anywhere else.

Source Code and Errata

You can and should download the source code for this book from the Apress web site at `www.apress.com`. Feel free to use it in your own projects, whether personal or commercial. We'll post any corrections to code as they're uncovered. We'll also post book corrections in the errata section.

How to Contact Us

We'd love to hear from you, whether it's questions, concerns, better ways of doing things, or triumphant announcements of your Core Data apps landing on the App Store. You can find us here:

Michael Privat
E-mail: `mprivat@mac.com`
Twitter: `@michaelprivat`
Blog: `http://michaelprivat.com`

Rob Warner
E-mail: `rwarner@grailbox.com`
Twitter: `@hoop33`
Blog: `http://grailbox.com`

Chapter 1

Getting Started

If you misread this book's title, thought it discussed and deciphered core dumps, and hope it will help you debug a nasty application crash, you got the wrong book. Get a debugger, memory tools, and an appointment with the optometrist. Otherwise, you bought, borrowed, burglarized, or acquired this book somehow because you want to better understand and implement Core Data in your iOS applications. You got the right book.

You might read these words from a paper book, stout and sturdy and smelling faintly of binding glue. You might digitally flip through these pages on a nook, iPad, Kindle, Sony Reader, Kobo eReader, or some other electronic book reader. You might stare at a computer screen, whether on laptop, netbook, or monitor, reading a few words at a time while telling yourself to ignore your Twitter feed rolling CNN-like along the screen's edge. As you read, you know that not only can you stop at any time but that you can resume at any time. Any time you want to read this book, you can pick it up. If you marked the spot where you were last reading, you can even start from where you last stopped. We take this for granted with books.

Users take it for granted with applications.

Users expect to find their data each time they launch their applications. Apple's Core Data framework helps you ensure that they will. This chapter introduces you to Core Data, explaining what it is, how it came to be, and how to build simple Core Data–based applications for iOS. This book walks through the simplicity and complexities of Core Data. Use the information in the book to create applications that store and retrieve data reliably and efficiently so that users can depend on their data. Code carefully, though—you don't want to write buggy code and have to deal with nasty application crashes.

What Is Core Data?

When people use computers, they expect to preserve any progress they make toward completing their tasks. Saving progress, essential to office software, code editors, and games involving small plumbers, is what programmers call *persistence*. Most software requires persistence, or the ability to store and retrieve data, so that users don't have to reenter all their data each time they use their applications. Some software can survive without any data storage or retrieval; calculators, carpenter's levels, and apps that make annoying or obscene sounds spring to mind. Most useful applications, however, preserve some state, whether configuration-oriented data, progress toward achieving some goal, or mounds of related data that users create and care about. Understanding how to persist data to iDevices is critical to most useful iOS development.

Apple's Core Data provides a versatile persistence framework. Core Data isn't the only data storage option, nor is it necessarily the best option in all scenarios, but it fits well with the rest of the Cocoa Touch development framework and maps well to objects. Core Data hides most of the complexities of data storage and allows you to focus on what makes your application fun, unique, or usable.

Although Core Data can store data in a relational database (such as SQLite), it is not a database engine. It doesn't even have to use a relational database to store its data. Though Core Data provides an entity-relationship diagramming tool, it is not a data modeler. It isn't a data access layer like Hibernate, though it provides much of the same object-relational mapping functionality. Instead, Core Data wraps the best of all these tools into a data management framework that allows you to work with entities, attributes, and relationships in a way that resembles the object graphs you're used to working with in normal object-oriented programming.

Early iPhone programmers didn't have the power of the Core Data framework to store and retrieve data. The next section shows you the history behind persistence in iOS.

History of Persistence in iOS

Core Data evolved from a NeXT technology called Enterprise Objects Framework (EOF) by way of WebObjects, another NeXT technology that still powers parts of Apple's web site. It debuted in 2005 as part of Mac OS X 10.4 ("Tiger"), but didn't appear on iPhones until version 3.0 of the SDK, released in June 2009. Before Core Data, iPhone developers had the following options in terms of persistence:

- Use property lists, which contain nested lists of key/value pairs of various data types.
- Serialize objects to files using the SDK's NSCoding protocol.
- Take advantage of the iPhone's support for the relational database SQLite.
- Persist data to the Internet cloud.

Developers used all these mechanisms for data storage as they built the first wave of applications that flooded Apple's App Store. Each one of these storage options remains viable, and developers continue to employ them as they build newer applications using newer SDK versions.

None of these options, however, compares favorably to the power, ease of use, and Cocoa-fitness of Core Data. Despite the invention of frameworks like FMDatabase or

ActiveRecord to make dealing with persistence on iOS easier in the pre–Core Data days, developers gratefully leapt to Core Data when it became available.

Although Core Data might not solve all persistence problems best and you might solve some of your persistence scenarios using other means like the options listed earlier, you'll turn to Core Data more often than not. As you work through this book and learn the problems that Core Data solves and how elegantly it solves them, you'll likely use Core Data any time you can. As new persistence opportunities arise, you won't ask yourself, "Should I use Core Data for this?" but rather, "Is there any reason *not* to use Core Data?"

The next section shows you how to build a basic Core Data application using Xcode's project templates. Even if you've already generated an Xcode Core Data project and know all the buttons and check boxes to click, don't skip the next section. It explains the Core Data–related sections of code that the templates generate and forms a base of understanding on which the rest of the book builds.

Creating a Basic Core Data Application

The many facets, classes, and nuances of Core Data merit artful analysis and deep discussions to teach you all you need to know to gain mastery of Core Data's complexities. Building a practical foundation to support the theory, however, is just as essential to mastery. This section builds a simple Core Data–based application using one of Xcode's built-in templates and then dissects the most important parts of its Core Data–related code to show what they do and how they interact. At the end of this section, you will understand how this application interacts with Core Data to store and retrieve data.

Understanding the Core Data Components

Before building this section's basic Core Data application, you should have a high-level understanding of the components of Core Data. Figure 1–1 illustrates the key elements of the application you will build in this section. Review this figure for a bird's-eye view of what this application accomplishes, where all its pieces fit, and why you need them.

As a user of Core Data, you should never interact directly with the underlying persistent store. One of the fundamental principles of Core Data is that the persistent store should be abstracted from the user. A key advantage of that is the ability to seamlessly change the backing store in the future without having to modify the rest of your code. You should try to picture Core Data as a framework that manages the persistence of objects rather than thinking about databases. Not surprisingly, the objects managed by the framework must extend `NSManagedObject` and are typically referred to as, well, managed objects. Don't think, though, that the lack of imagination in the naming conventions for the components of Core Data reveals an unimaginative or mundane framework. In fact, Core Data does an excellent job at keeping all the object graph interdependencies, optimizations, and caching in a predictable state so that you don't have to worry about it. If you have ever tried to build your own object management framework, you understand all the intricacies of the problem Core Data solves for you.

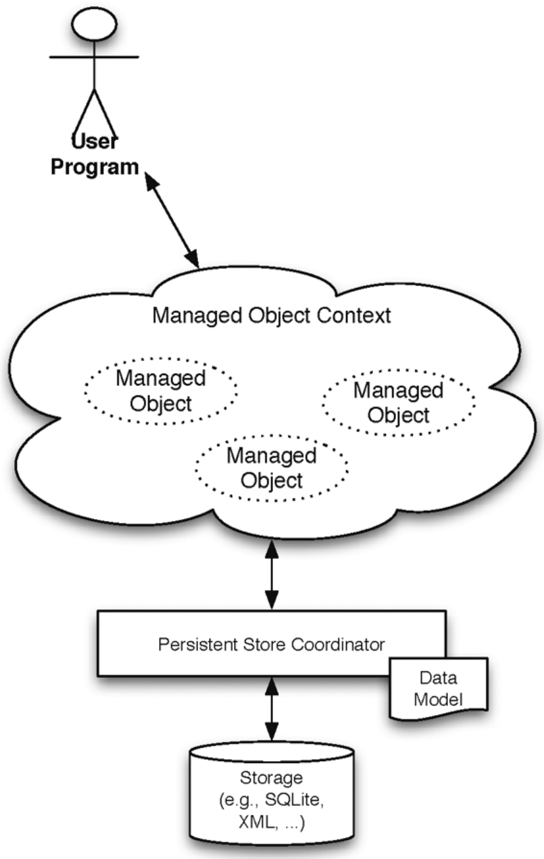
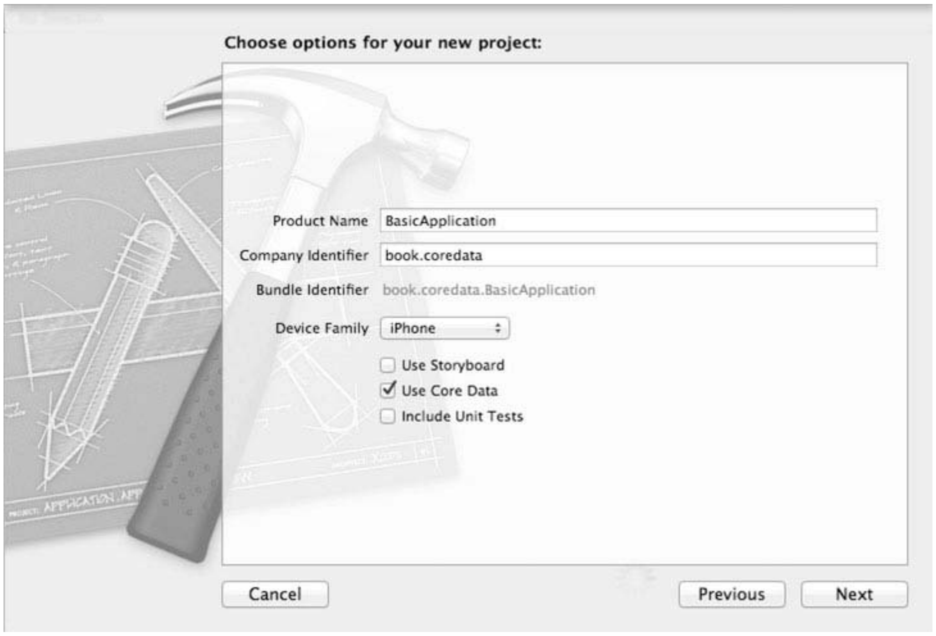


Figure 1–1. *Overview of Core Data's components*

Much like we need a livable environment to subsist, managed objects must live within an environment that's livable for them, usually referred to as a *managed object context*, or simply *context*. The context keeps track of the states of not only the object you are altering but also all the objects that depend on it or that it depends on. The `NSManagedObjectContext` object in your application provides the context and is the key property that your code must always be able to access. You typically accomplish exposing your `NSManagedObjectContext` object to your application by having your application delegate initialize it and expose it as one of its properties. Your application context will often give the `NSManagedObjectContext` object to the main view controller as well. Without the context, you will not be able to interact with Core Data.

Creating a New Project

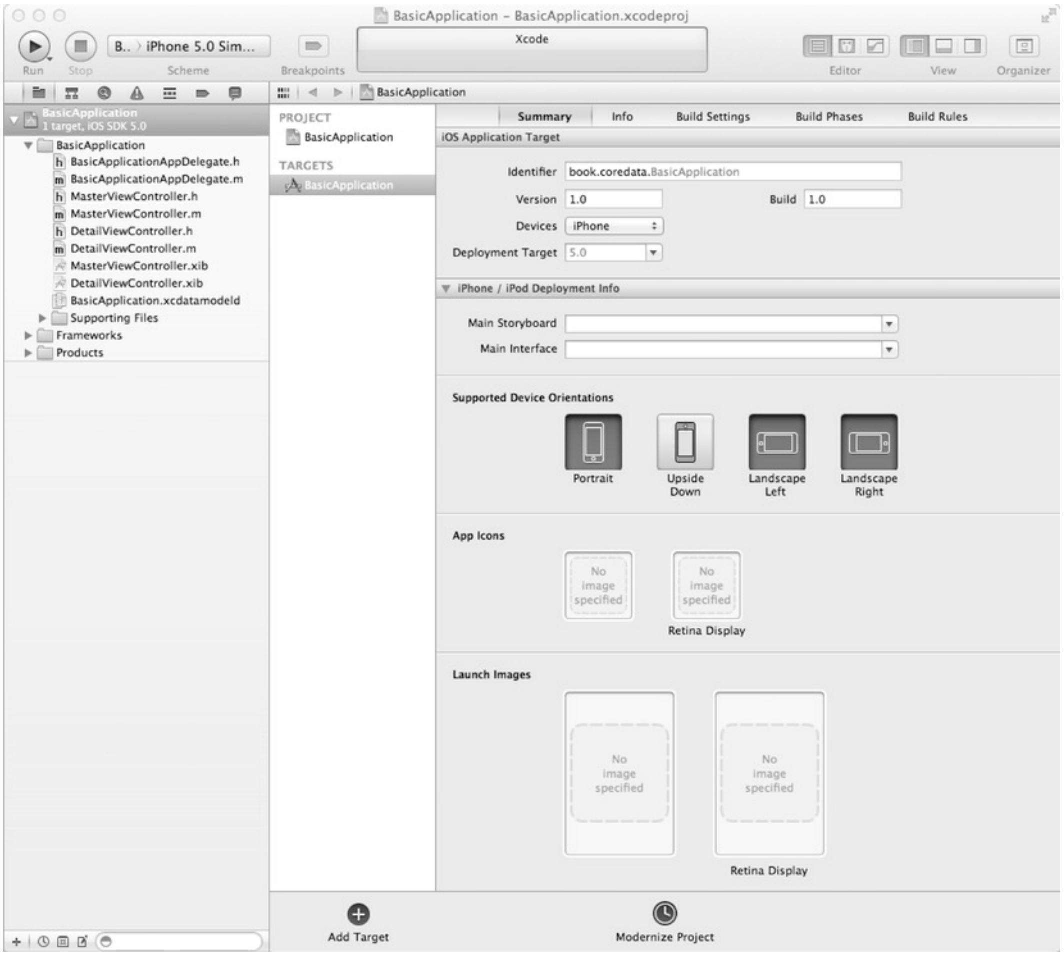
To begin, launch Xcode, and create a new project by selecting **File ➤ New ➤ New Project** from the menu. Note that you can also create a new project by pressing ⇧+⌘+N. From the list of application templates, select the Application item under iOS on the left, and pick Master-Detail Application on the right. Click Next, and on the next screen type **BasicApplication** in the Product Name field, **book.coredata** in the Company Identifier field, uncheck Use Storyboard and check Use Core Data. See Figure 1–2. Click the Next button, choose the parent directory where Xcode will create the BasicApplication directory and project, and click Create. Xcode creates your project, generates the project's files, and opens its IDE window with all the files it generated, as Figure 1–3 shows.


Choose options for your new project:

Product Name: BasicApplication
Company Identifier: book.coredata
Bundle Identifier: book.coredata.BasicApplication
Device Family: iPhone

- [ ] Use Storyboard
- [x] Use Core Data
- [ ] Include Unit Tests

Figure 1–2. *Creating a new project with Core Data*



Identifier: book.coredata.BasicApplication
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 5.0
Main Storyboard:
Main Interface:

Figure 1–3. *Xcode showing your new project*

Running Your New Project

Before digging into the code, run it to see what it does. Launch the application by clicking the Run button. The iPhone Simulator opens, and the application presents the navigation-based interface shown in Figure 1–4, with a table view occupying the bulk of the screen, an Edit button in the top-left corner, and the conventional Add button, denoted by a plus sign, in the upper-right corner. The application's table shows an empty list indicating that the application isn't aware of any events, which is what the generated Xcode Core Data project stores. Create a new event stamped with the current time by clicking the plus button in the top-right corner of the application.

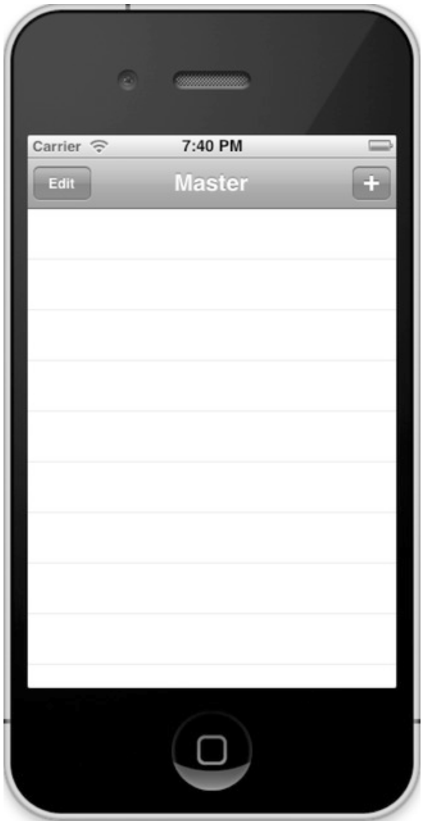


Figure 1–4. *The basic application with a blank screen*

Now, stop the application by clicking the Stop button in the Xcode IDE. If the application hadn't used Core Data persistence, it would have lost the event you just created as it exited. Maintaining a list of events with this application and no persistence would be a Sisyphean task—you'd have to re-create the events each time you launched the application. Because the application uses persistence, however, it stored the event you created using the Core Data framework. Relaunching the application shows that the event is still there, as Figure 1–5 demonstrates.

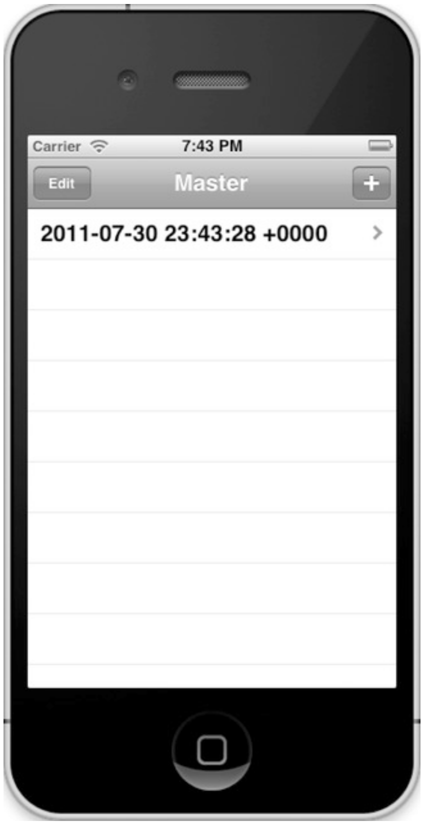


Figure 1–5. *The basic application with a persisted event*

Understanding the Application's Components

The anatomy of the application is relatively simple. It has a data model that describes the entities in the data store, a view controller that facilitates interactions between the view and the data store, and an application delegate that helps initialize and launch the application. Figure 1–6 shows the classes involved and how they relate to each other.

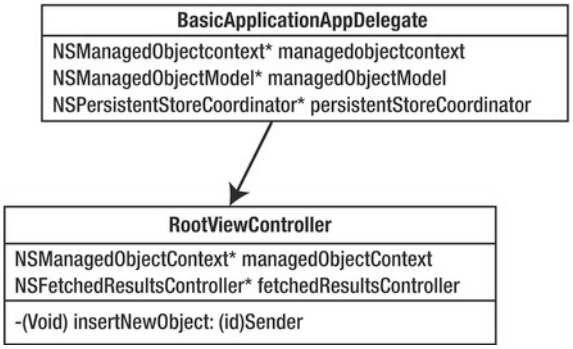

BasicApplicationAppDelegate
NSManagedObjectcontext* managedobjectcontext
NSManagedObjectModel* managedObjectModel
NSPersistentStoreCoordinator* persistentStoreCoordinator
RootViewController
NSManagedObjectContext* managedObjectContext
NSFetchedResultsController* fetchedResultsController
-(Void) insertNewObject: (id)Sender

Figure 1–6. *Classes involved in the BasicApplication example*

Note how the `MasterViewController` class, which is in charge of managing the user interface, has a handle to the managed object context so that it can interact with Core Data. As you go through the code, you'll see that the `MasterViewController` class obtains the managed object context from the application delegate. This happens in the controller's `initWithNibName:bundle:` method, shown here:

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
  self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
  if (self) {
    self.title = NSLocalizedString(@"Master", @"Master");
    id delegate = [[UIApplication sharedApplication] delegate];
    self.managedObjectContext = [delegate managedObjectContext];
  }
  return self;
}
```

The entry called `BasicApplication.xcdatamodeld`, which is actually a directory on the file system, contains the data model, `BasicApplication.xcdatamodel`. The data model is central to every Core Data application. This particular data model defines only one entity, named `Event`, for the application. `Events` are defined as entities that contain only one attribute named `timeStamp` of type `Date`, as shown in Figure 1–7.

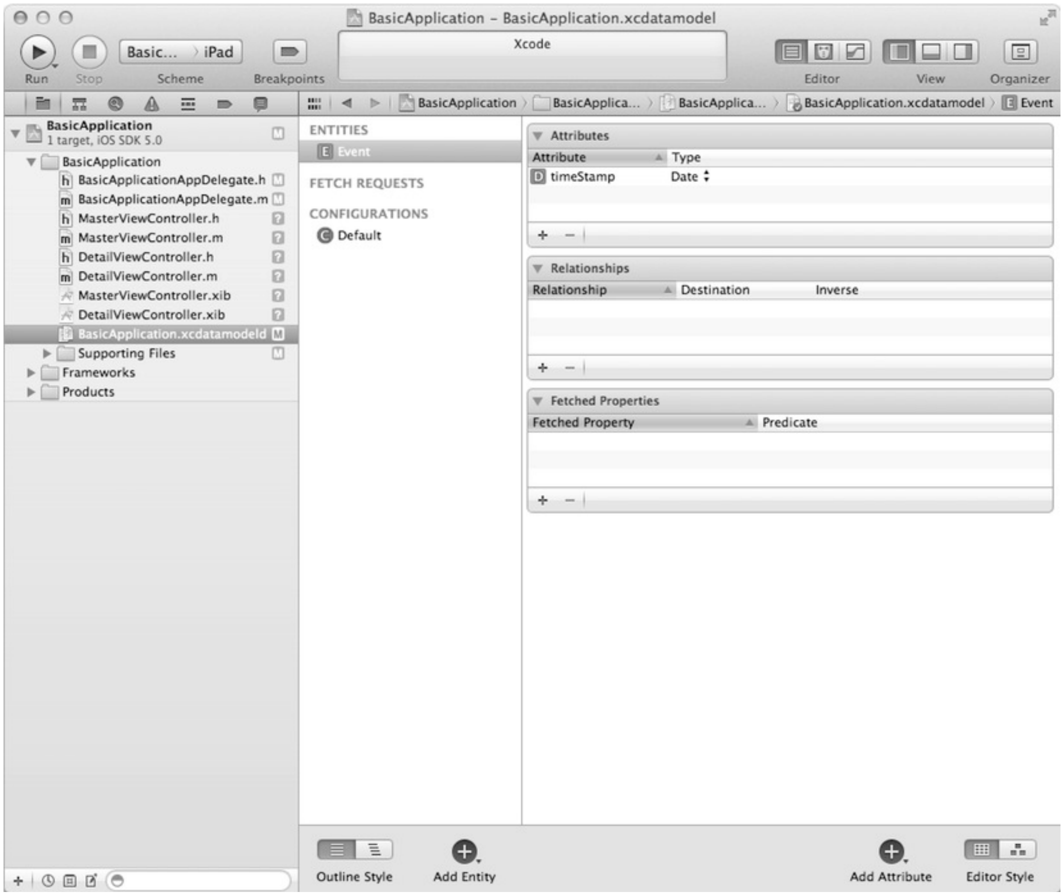


Figure 1–7. *The Xcode-generated data model*

The Event entity is of type NSManagedObject, which is the basic type for all entities managed by Core Data. Chapter 2 explains the NSManagedObject type in more detail.

Fetching Results

The next class of interest is the MasterViewController. Opening its header file (MasterViewController.h) reveals two properties:

```
@property (strong, nonatomic) NSFetchedResultsController *fetchedResultsController;
@property (strong, nonatomic) NSManagedObjectContext *managedObjectContext;
```

These properties are defined using the same syntax as the definitions of any Objective-C class properties. The NSFetchedResultsController is a type of controller provided by the Core Data framework that helps manage results from queries. NSManagedObjectContext is a handle to the application's persistent store that provides a context, or environment, in which the managed objects can exist.

The implementation of the MasterViewController, found in MasterViewController.m, shows how to interact with the Core Data framework to store and retrieve data. The MasterViewController implementation provides an explicit getter for the fetchedResultsController property that preconfigures it to fetch data from the data store.

The first step in creating the fetch controller consists of creating a request that will retrieve Event entities, as shown in this code from the fetchedResultsController accessor:

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
 inManagedObjectContext:self.managedObjectContext];
[fetchRequest setEntity:entity];
```

The result of the request can be ordered using the sort descriptor from the Cocoa Foundation framework. The sort descriptor defines the field to use for sorting and whether the sort is ascending or descending. In this case, you sort by descending chronological order, like so:

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:
@"timeStamp" ascending:NO];
NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];
[fetchRequest setSortDescriptors:sortDescriptors];
```

Once you define the request, you can use it to construct the fetch controller. Because the MasterViewController implements NSFetchedResultsControllerDelegate, it can be set as the NSFetchedResultsController's delegate so that it is automatically notified as the result set changes and so that it updates its view appropriately. You could get the same results by invoking the executeFetchRequest of the managed object context, but you would not benefit from the other advantages that come from using the NSFetchedResultsController such as the seamless integration with the UITableView, as you'll see later in this section and in Chapter 9. Here is the code that constructs the fetch controller:

```
NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController
 alloc] initWithFetchRequest:fetchRequest managedObjectContext:
self.managedObjectContext sectionNameKeyPath:nil cacheName:@"Master"];
aFetchedResultsController.delegate = self;
self.fetchedResultsController = aFetchedResultsController;
```

NOTE: You may have noticed that the initWithFetchRequest shown earlier uses a parameter called cacheName. You could pass nil for the cacheName parameter to prevent caching, but naming a cache indicates to Core Data to check for a cache with a name matching the passed name and see whether it already contains the same fetch request definition. If it does find a match, it will reuse the cached results. If it finds a cache entry by that name but the request doesn't match, then it is deleted. If it doesn't find it at all, then the request is executed, and the cache entry is created for the next time. This is obviously an optimization that aims to prevent executing the same request over and over. Core Data manages its caches intelligently so that if the results are updated by another call, the cache is removed if affected.

Finally, you tell the controller to execute its query to start retrieving results. To do this, use the `performFetch` method.

```
NSError *error = nil;
if (![self.fetchedResultsController performFetch:&error]) {
  NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
  abort();
}
```

You can see the entire getter method for `fetchedResultsController` in Listing 1–1.

Listing 1–1. *The Entire Getter Method for fetchedResultsController*

```
- (NSFetchedResultsController *)fetchedResultsController
{
  if (__fetchedResultsController != nil)
  {
    return __fetchedResultsController;
  }

  /*
   Set up the fetched results controller.
   */
  // Create the fetch request for the entity.
  NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
  // Edit the entity name as appropriate.
  NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
inManagedObjectContext:self.managedObjectContext];
  [fetchRequest setEntity:entity];

  // Set the batch size to a suitable number.
  [fetchRequest setFetchBatchSize:20];

  // Edit the sort key as appropriate.
  NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"timeStamp"
ascending:NO];
  NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];

  [fetchRequest setSortDescriptors:sortDescriptors];

  // Edit the section name key path and cache name if appropriate.
  // nil for section name key path means "no sections".
  NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController
alloc] initWithFetchRequest:fetchRequest managedObjectContext:self.managedObjectContext
sectionNameKeyPath:nil cacheName:@"Master"];
  aFetchedResultsController.delegate = self;
  self.fetchedResultsController = aFetchedResultsController;

        NSError *error = nil;
        if (![self.fetchedResultsController performFetch:&error])
  {
    /*
     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. If it is not possible to recover from the error, display an alert panel
that instructs the user to quit the application by pressing the Home button.
```

```
     */
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
        }

  return __fetchedResultsController;
}
```

NSFetchedResultsController behaves as a collection of managed objects, similar to an NSArray, which makes it easy to use. In fact, it exposes a read-only property called fetchedObjects that is of type NSArray to make things even easier to access the objects it fetches. The MasterViewController class, which also extends UITableViewController, demonstrates just how suited the NSFetchedResultsController is to manage the table's content.

Inserting New Objects

A quick glance at the insertNewObject: method shows how new events (the managed objects) are created and added to the persistent store. Managed objects are defined by the entity description from the data model and can live only within a context. The first step is to get a hold of the current context as well as the entity definition. In this case, instead of explicitly naming the entity, you reuse the entity definitions that are attached to the fetched results controller:

```
NSManagedObjectContext *context = [self.fetchedResultsController managedObjectContext];
NSEntityDescription *entity = [[self.fetchedResultsController fetchRequest] entity];
```

Now that you've gathered all the elements needed to bring the new managed object to existence, you create the Event object and set its timeStamp value.

```
NSManagedObject *newManagedObject = [NSEntityDescription
 insertNewObjectForEntityForName:[entity name] inManagedObjectContext:context];
[newManagedObject setValue:[NSDate date] forKey:@"timeStamp"];
```

The last step of the process is to tell Core Data to save changes to its context. The obvious change is the object you just created, but keep in mind that calling the save. method will also affect any other unsaved changes to the context.

```
NSError *error = nil;
if (![context save:&error]) {
  NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
  abort();
}
```

The complete method for inserting the new Event object is shown in Listing 1–2.

Listing 1–2. *The Complete Method for Inserting the New Event Object*

```
- (void)insertNewObject {
  // Create a new instance of the entity managed by the fetched results controller.
  NSManagedObjectContext *context = [self.fetchedResultsController
managedObjectContext];
  NSEntityDescription *entity = [[self.fetchedResultsController fetchRequest] entity];
  NSManagedObject *newManagedObject = [NSEntityDescription
insertNewObjectForEntityForName:[entity name] inManagedObjectContext:context];
```

```
  // If appropriate, configure the new managed object.
  // Normally you should use accessor methods, but using KVC here avoids the need to add
a custom class to the template.
  [newManagedObject setValue:[NSDate date] forKey:@"timeStamp"];

  // Save the context.
  NSError *error = nil;
  if (![context save:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
  }
}
```

Initializing the Managed Context

Obviously, none of this can happen without initializing the managed context first. This is the role of the application delegate. In a Core Data–enabled application, the delegate must expose three properties.

```
@property (readonly, strong, nonatomic) NSManagedObjectContext *managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel *managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
*persistentStoreCoordinator;
```

Note that they are all marked as read-only, which prevents any other component in the application from setting them directly. A closer look at `BasicApplicationAppDelegate.m` shows that all three properties have explicit getter methods.

First, the managed object model is derived from the data model (`BasicApplication.xcdatamodel`) and loaded.

```
- (NSManagedObjectModel *)managedObjectModel
{
    if (__managedObjectModel != nil)
    {
        return __managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"BasicApplication"
withExtension:@"momd"];
    __managedObjectModel = [[NSManagedObjectModel alloc]
initWithContentsOfURL:modelURL];
    return __managedObjectModel;
}
```

Then a persistent store is created to support the model. In this case, as well as in most Core Data scenarios, the persistent store is backed by a SQLite database. The managed object model is a logical representation of the data store, while the persistent store is the materialization of that data store.

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
  if (__persistentStoreCoordinator != nil) {
    return __persistentStoreCoordinator;
  }
  NSURL *storeURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"BasicApplication.sqlite"];

  NSError *error = nil;
  __persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:[self managedObjectModel]];
  if (![__persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:nil URL:storeURL options:nil error:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
  }

  return __persistentStoreCoordinator;
}
```

Finally, the managed object context is created.

```
- (NSManagedObjectContext *)managedObjectContext {
  if (__managedObjectContext != nil) {
    return __managedObjectContext;
  }

  NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
  if (coordinator != nil) {
    __managedObjectContext = [[NSManagedObjectContext alloc] init];
    [__managedObjectContext setPersistentStoreCoordinator:coordinator];
  }
  return __managedObjectContext;
}
```

The context is used throughout the application as the single interface with the Core Data framework and the persistent store, as Figure 1–8 demonstrates.

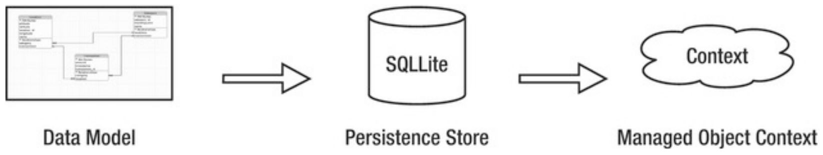


Figure 1–8. *Core Data initialization sequence*

Lastly, everything is put in motion when the application delegate's `application:didFinishLaunchingWithOptions:` method is called.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.

  MasterViewController *controller = [[MasterViewController alloc]
initWithNibName:@"MasterViewController" bundle:nil];
  self.navigationController = [[UINavigationController alloc]
initWithRootViewController:controller];
  self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];
    return YES;
}
```