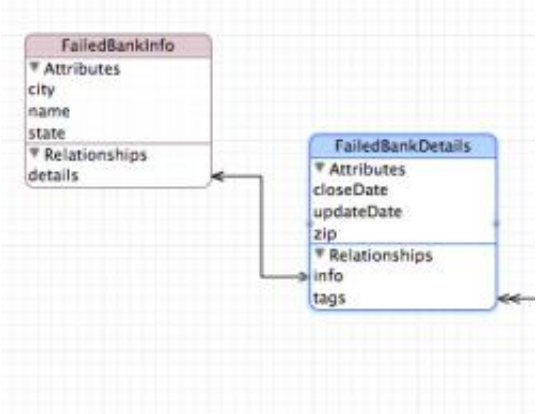
Core Data on iOS 5 Tutorial: How To Work with Relations and Predicates

笔记本：	陈金声的电脑
创建时间：	2012/10/29 7:52
标签：	iOS-Dev
URL：	http://www.raywenderlich.com/14742/core-data-on-ios-5-tutorial-how-to-work-with-relations-and-predicates

5 JULY 2012

Core Data on iOS 5 Tutorial: How To Work with Relations and Predicates

This is a post by iOS Tutorial Team member Cesare Rocchi, a UX designer and developer specializing in web and mobile applications.



Learn how to use relationships and predicates with Core Data!

Good news – by popular request, we now have a 4th part to our Core Data tutorial series! :]

In the first part of the series, you learned how to visually build a data model and show it in a table view.

In the second part of the series, you saw how to import data from an existing database to populate an application with information.

The third part of the series, you learned how to use NSFetchedResultsController to retrieve data from the object graph.

Now in Part 4, you'll learn how to deal with predicates and relationships. You'll start by modifying the project to allow editing objects. Then you'll introduce relationships into the project and learn how to build specific queries by using NSPredicate.

If you're already familiar with the basics of Core Data but just want to learn more about relationships and predicates, don't

worry you can still follow along. Just take a few moment to look over the starter project first.

The starter project will be the finished project as it stood at the end of Part 3 of the series. You can grab it here.

Now let's continue our lovely relationship with Core Data!

Getting Started: Editing, Not Just for Writers

As a reminder, we left off in part 3 with a simple Core Data app that shows a list of banks that have failed in the US. A bit too long of a list for comfort! :]

However, the list was read-only – no editing! So before we go any further, let's add editing into the app. This will make the app more functional, all while teaching you about relations and predicates in Core Data.

More precisely, you'll introduce the functionality to add a bank to the list and to edit banks already stored in the database. Then you'll make it possible to search the list for banks that meet certain criteria.

If you just downloaded the project, extract the ZIP file to a location of your choice. Open the project in Xcode.

To start with a clean slate, you'll get rid of the procedure that imports the sqlite database. This way, our list of banks will be empty and we can add new ones with our soon-to-come editing capabilities.

So go to **FBCDAppDelegate.m**, go to **persistentStoreCoordinator** and delete the following code.

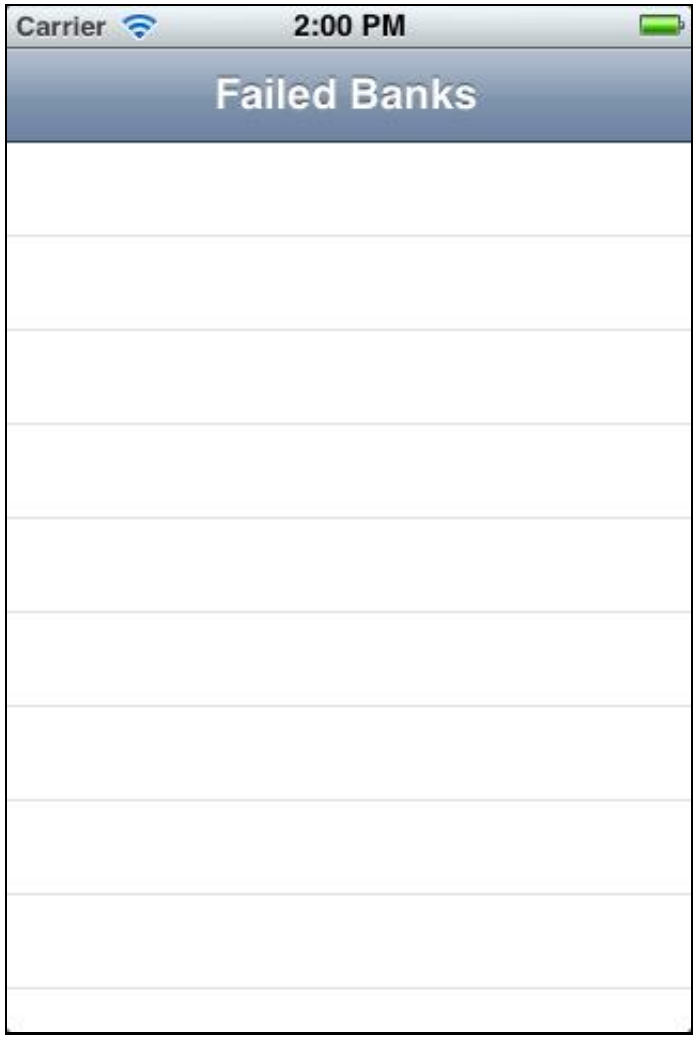
```
    if (![[NSFileManager defaultManager] fileExistsAtPath:[storeURL path]]) {
        NSURL *preloadURL = [NSURL fileURLWithPath:[[NSBundle mainBundle] pathForResource:@"CoreDataTutorial2" ofType:@

        NSError* err = nil;

        if (![[NSFileManager defaultManager] copyItemAtURL:preloadURL toURL:storeURL error:&err]) {
            NSLog(@"Oops, could copy preloaded data");
        }
    }
```

In the same file, go to **application:didFinishLaunchingWithOptions:** and delete the following:

```
    NSManagedObjectContext *context = [self managedObjectContext];
    FailedBankInfo *failedBankInfo = [NSEntityDescription
                                      insertNewObjectForEntityForName:@"FailedBankInfo"
                                      inManagedObjectContext:context];
    failedBankInfo.name = @"Test Bank";
    failedBankInfo.city = @"Testville";
    failedBankInfo.state = @"Testland";
    FailedBankDetails *failedBankDetails = [NSEntityDescription
                                            insertNewObjectForEntityForName:@"FailedBankDetails"
                                            inManagedObjectContext:context];
    failedBankDetails.closeDate = [NSDate date];
    failedBankDetails.updateDate = [NSDate date];
    failedBankDetails.zip = [NSNumber numberWithInt:12345];
    failedBankDetails.info = failedBankInfo;
    failedBankInfo.details = failedBankDetails;
    NSError *error;
    if (![context save:&error]) {
        NSLog(@"Whoops, couldn't save: %@", [error localizedDescription]);
    }

    // Test listing all FailedBankInfos from the store
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"FailedBankInfo"
                                              inManagedObjectContext:context];
    [fetchRequest setEntity:entity];
    NSArray *fetchedObjects = [context executeFetchRequest:fetchRequest error:&error];
    for (FailedBankInfo *info in fetchedObjects) {
        NSLog(@"Name: %@", info.name);
        FailedBankDetails *details = info.details;
        NSLog(@"Zip: %@", details.zip);
    }
```

And with those few clicks and keystrokes, the project is clean! It no longer imports any data from a batch procedure or inserts data when the application is launched.

Build and run, and you should see an empty table view, as follows:



Now you can set about making it possible for users to add data as they see fit.

Now add two buttons to FBCDMasterViewController. One will be used to add a new bank, and the other will show the search view. In FBCDMasterViewController.m, add the following code to the bottom of viewDidLoad.

```
    self.navigationItem.leftBarButtonItem = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemA
                                                                                          target:self
                                                                                          action:@selector(addBank)];

    self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItem
                                                                                           target:self
                                                                                           action:@selector(showSearch)
```

Let's focus on `addBank` first. Add the method to the end of `FBCDMasterViewController.m` (but before the final @end):

```
-(void)addBank {

    FailedBankInfo *failedBankInfo = (FailedBankInfo *)[NSEntityDescription insertNewObjectForEntityForName:@"FailedBan
                                                                                        inManagedObjectContext:managedObje
    failedBankInfo.name = @"Test Bank";
    failedBankInfo.city = @"Testville";
    failedBankInfo.state = @"Testland";

    FailedBankDetails *failedBankDetails = [NSEntityDescription insertNewObjectForEntityForName:@"FailedBankDetails"
                                                                                 inManagedObjectContext:managedObjectContext];
    failedBankDetails.closeDate = [NSDate date];
    failedBankDetails.updateDate = [NSDate date];
    failedBankDetails.zip = [NSNumber numberWithInt:123];
    failedBankDetails.info = failedBankInfo;
    failedBankInfo.details = failedBankDetails;

    NSError *error = nil;
    if (![managedObjectContext save:&error]) {
        NSLog(@"Error in adding a new bank %@, %@", error, [error userInfo]);
        abort();
    }

}
```

The above code is pretty similar to the code you deleted before. You create an instance of FailedBankInfo and you populate the properties with values. One of the properties is an instance of FailedBankDetails, which you set as the "details" property.

Finally, you save the context to make sure the insertion is committed to the database. If you run the application now, you should notice that the table view gets updated correctly with the new instance without requiring a call to reloadData. How come?

This is due to the these functions, both inherited from previous versions of the project:

- **controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:** This takes care of four possible changes to the table view: insertions, deletions, updates and moves.
- **controllerWillChangeContent:** This simply "alerts" the controller of upcoming changes via the fetched results controller.

Deleting Banks

Now that you've got the ability to add a bank, let's add deletion as well!

You can enable that by adding the swipe-to-delete functionality, built-in to table views. You just need to add two methods.

The first new method simply indicates which cells in the table are editable. You can either add the following code below **tableView:cellForRowAtIndexPath:**, or you can uncomment the pre-existing commented-out block of code for the method:

```
-(BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath *)indexPath {
    return YES;
}
```

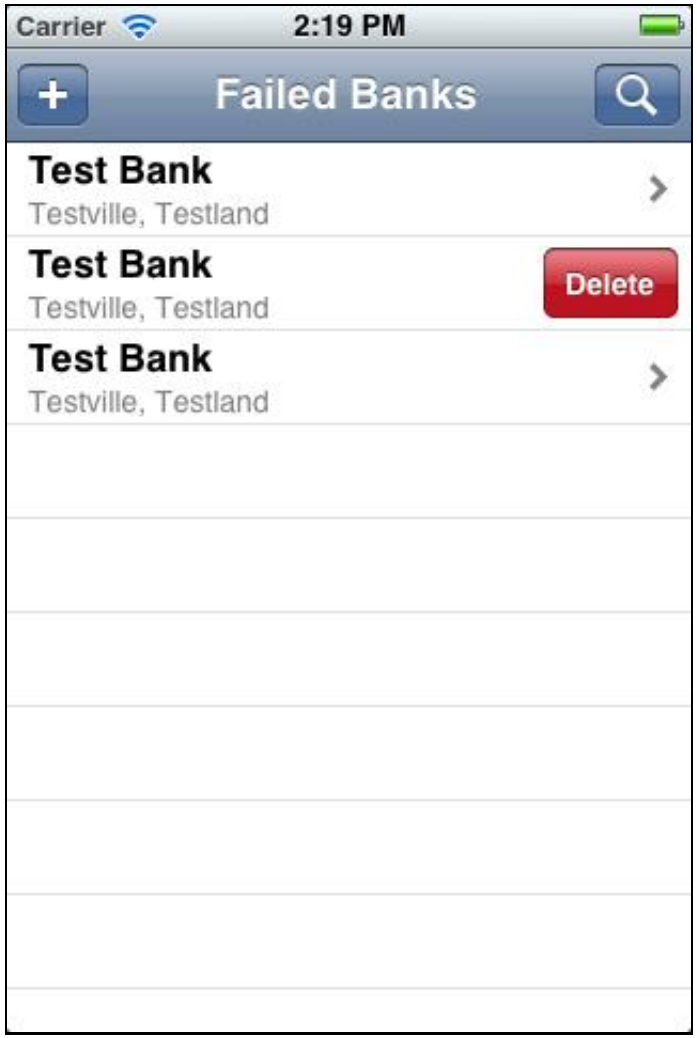
The second new method, also to be added to **FBCDMasterViewController.m**, is **tableView:commitEditingStyle:forRowAtIndexPath:**. Again, there is a commented-out section of code for this, but instead of using it, replace that code with the following:

```
-(void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle forRowAtIndexPat
    if (editingStyle == UITableViewCellEditingStyleDelete) {

        [managedObjectContext deleteObject:[self.fetchedResultsController objectAtIndexPath:indexPath]];
```

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

As before, there's no need to refresh the table view! It's all handled by the fetched results controller, which notifies the controller that changes have occured.

Build and run, add a few banks, and swipe one of the cells to show the delete button. Tap the button, and you'll see the record deleted and the table view refreshed!

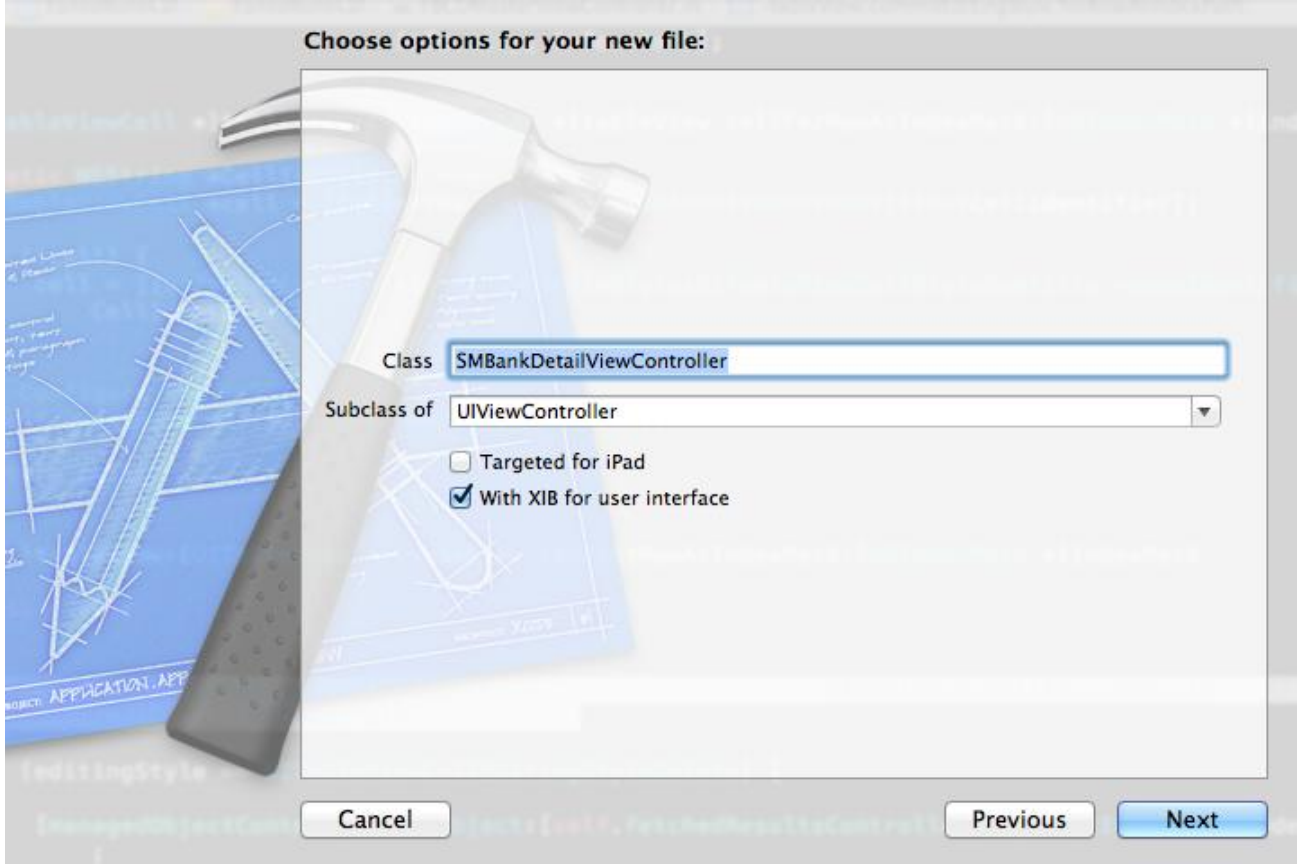


Editing Banks

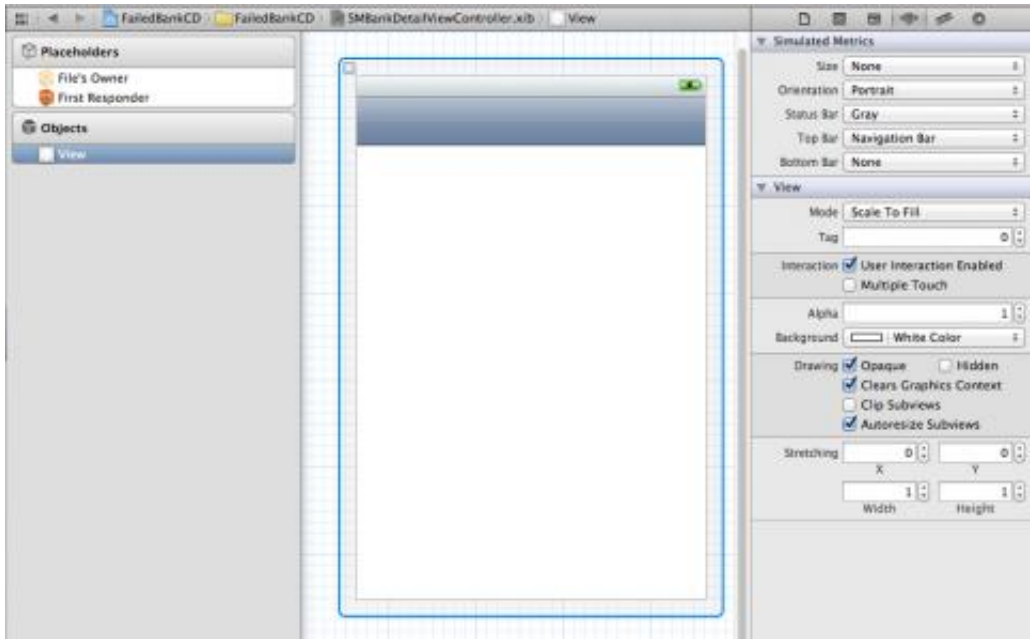
At this point, new banks are added using "static" content. Now you'll incorporate the ability to edit the details of a bank. To

handle this, you'll build a new view controller.

Right-click on the project root and select New File/Cocoa Touch/Objective-C Class. Name the controller SMBankDetailViewController and make it a subclass of UIViewController. Also make sure that a XIB is generated.



Now open SMBankDetailViewController.xib to add a few components. The view will be pushed by a view controller, so you might want to visualize the space taken by a navigation bar. With the view selected, tap the fourth icon in the inspector (the right sidebar) and set the top bar to "Navigation Bar."



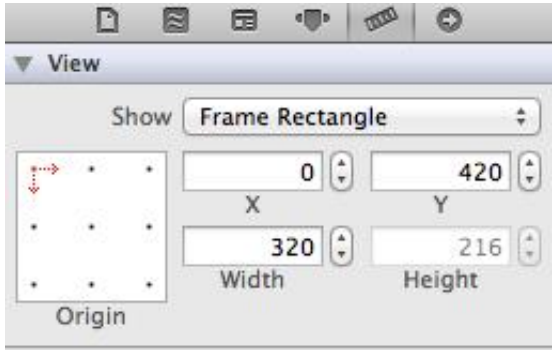
Then drag four text fields and two labels to the view, and lay them out as in the following screenshot:


Name
City
Zip
State
Close date
Tags

You might want to edit the placeholder text of the input fields so it's clear what's what – you can use the above screenshot as a reference. Later on in this tutorial, you'll set the tags on the labels as well.

While you're at it, add a date picker component. This will be displayed when necessary via code to edit dates.

For the moment, place the picker outside of the visible area of the view. The Y of the picker should be set to 420. With the picker selected, switch to the Size Inspector tab on the right sidebar and set its position as follows:



Now it's time to write some code to display the new view. In SMBankDetailViewController.h, add the following import statements below the existing #import line:

```
#import "FailedBankInfo.h"
#import "FailedBankDetails.h"
```

Then add the following properties and methods before the final @end:

```
@property (nonatomic, strong) FailedBankInfo *bankInfo;
@property (nonatomic, weak) IBOutlet UITextField *nameField;
@property (nonatomic, weak) IBOutlet UITextField *cityField;
@property (nonatomic, weak) IBOutlet UITextField *zipField;
@property (nonatomic, weak) IBOutlet UITextField *stateField;
@property (nonatomic, weak) IBOutlet UILabel *tagsLabel;
@property (nonatomic, weak) IBOutlet UILabel *dateLabel;
@property (nonatomic, weak) IBOutlet UIDatePicker *datePicker;
```

```
-(id)initWithBankInfo:(FailedBankInfo *) info;
```

Next, in SMBankDetailViewController.xib, hook up each outlet you defined with its corresponding component. You can do this by selecting the File's Owner (which is the SMBankDetailViewController class), switching to the Connections Inspector in the right sidebar, and dragging from each outlet to the relevant control on the view.

At the top of SMBankDetailViewController.m, right below the @implementation line, synthesize all the properties as follows:

```
@synthesize bankInfo = _bankInfo;
@synthesize nameField;
@synthesize cityField;
@synthesize zipField;
@synthesize stateField;
@synthesize tagsLabel;
@synthesize dateLabel;
@synthesize datePicker;
```

Also add two private methods to the class continuation above the @implementation line, as follows:

```
@interface SMBankDetailViewController ()
-(void)hidePicker;
-(void)showPicker;
@end
```

initWithBankInfo: is pretty simple: it assigns an instance of info to the view controller. Add it below the existing initWithNibName:bundle: method implementation:

```
-(id)initWithBankInfo:(FailedBankInfo *)info {
    if (self = [super init]) {
        _bankInfo = info;
    }
    return self;
}
```

Then add the following to the end of viewDidLoad to set a few parameters like the title, the right navigation item and a few

gesture recognizers:

```
    self.title = self.bankInfo.name;

    // 1 - setting the right button
    self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItem
        target:self action:@selector(saveBankInfo)];

    // 2 - setting interaction on date label
    self.dateLabel.userInteractionEnabled = YES;
    UITapGestureRecognizer *dateTapRecognizer = [[UITapGestureRecognizer alloc] initWithTarget:self
        action:@selector(dateTapped)];
    [self.dateLabel addGestureRecognizer:dateTapRecognizer];
    // 3 - set date picker handler
    [datePicker addTarget:self action:@selector(dateHasChanged:)
         forControlEvents:UIControlEventValueChanged];
```

In section 1, the right button on the navigation bar triggers a save action. The operation in this case is easy: set the values of the bank info as the values specified in the components, and save the context. Add the code for it to the end of the file (but before the final @end):

```
-(void)saveBankInfo {

    self.bankInfo.name = self.nameField.text;
    self.bankInfo.city = self.cityField.text;
    self.bankInfo.details.zip = [NSNumber numberWithInt:[self.zipField.text intValue]];
    self.bankInfo.state = self.stateField.text;

    NSError *error;
    if ([self.bankInfo.managedObjectContext hasChanges] && ![self.bankInfo.managedObjectContext save:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    [self.navigationController popViewControllerAnimated:YES];
```

```
}
```

The second tap detector (in section 2 above) calls the method `showPicker`. Add it to the end of the file:

```
-(void)dateTapped {
    [self showPicker];
}
```

The third selector (section 3) changes the value of the date label according to the selected value in the date picker. Add the necessary code again to the end of the file:

```
-(void)dateHasChanged:(id)sender {
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateStyle:NSDateFormatterMediumStyle];
    self.dateLabel.text = [formatter stringFromDate:self.datePicker.date];

}
```

`viewWillAppear:` sets the values of the text fields and labels according to the instance of bank info associated with the controller. Add the following code below `viewDidLoad`:

```
-(void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    // setting values of fields
    self.nameField.text = self.bankInfo.name;
    self.cityField.text = self.bankInfo.city;
    self.zipField.text = [self.bankInfo.details.zip stringValue];
    self.stateField.text = self.bankInfo.state;

    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateStyle:NSDateFormatterMediumStyle];
    self.dateLabel.text = [formatter stringFromDate:self.bankInfo.details.closeDate];
}
```

Finally, you're left with the implementation of the two private methods. Add the code for those to the end of the file:

```
-(void)showPicker {
    [self.zipField resignFirstResponder];
    [self.nameField resignFirstResponder];
    [self.stateField resignFirstResponder];
    [self.cityField resignFirstResponder];

    [UIView beginAnimations:@"SlideInPicker" context:nil];
        [UIView setAnimationDuration:0.5];
        self.datePicker.transform = CGAffineTransformMakeTranslation(0, -216);
        [UIView commitAnimations];
}

-(void)hidePicker {
    [UIView beginAnimations:@"SlideOutPicker" context:nil];
        [UIView setAnimationDuration:0.5];
        self.datePicker.transform = CGAffineTransformMakeTranslation(0, 216);
        [UIView commitAnimations];
}
```

The above code shows or hides the date picker, as necessary. Before showing the date picker, the first responder for all text fields is resigned, thus effectively dismissing the keyboard if it was visible.

To test your new view, you need to push it onto the navigation stack when a cell is tapped. In `FBCDMasterViewController.m`, add the following import statement at the top:

```
#import "SMBankDetailViewController.h"
```

Then replace the existing placeholder for `tableView:didSelectRowAtIndexPath:` with the following:

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    FailedBankInfo *info = [_fetchedResultsController objectAtIndexPath:indexPath];
    SMBankDetailViewController *detailViewController = [[SMBankDetailViewController alloc] initWithBankInfo:info];
    [self.navigationController pushViewController:detailViewController animated:YES];
}
```

Now you're ready to see the first big change to your app!

Run the app and create a few instances of banks. Each bank record is editable, including the close date. To save data, hit the save button; to discard changes, just tap the back button.

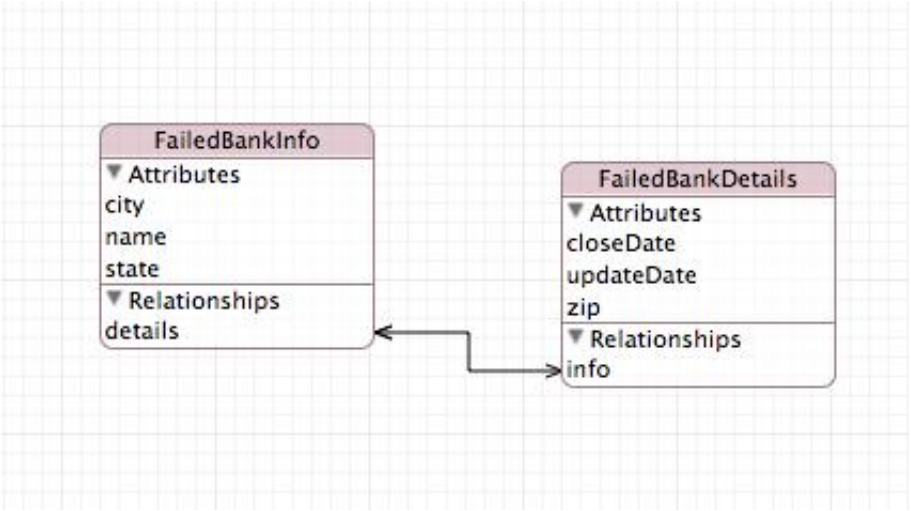
Notice that the date picker and the keyboard never obstruct each other. Changes are reflected in the list of banks with no need to refresh the table view. Pretty cool, huh?

Relationships, But Not the Dating Kind

In database terminology, a relationship is a "connection" between two entities. It's often translated into everyday language

using the verbs to have or to belong. Think of the classic example of employees and departments – an employee is said to belong to a department, and a department has employees.

In database modeling, relationships can be of three types: one-to-one, one-to-many and many-to-many. This property is usually referred to as **cardinality**. In the example from the previous section, there is already a relation modeled in Core Data: the one between FailedBankInfo and FailedBankDetails.



This is a one-to-one relationship: each info object can have exactly one details object associated with it and vice versa. The graphical view stresses this point by connecting the two entities with one single arrow line. In other words, these two only have eyes for each other. :]

Whenever you define a relationship, you have to specify the following:

1. **Name**: This is just a string identifying the name of the relation.
2. **Destination entity**: This is the target or the destination class of the relation. For example, the relationship that goes from an employee to the department can be called "department." In this case, the employee is called the source and the department is the destination.
3. **Cardinality**: The answer to the question: Is the destination a single object or not? If yes, the relation is of type to-one,

otherwise it is a to-many. Assuming that in your scenario an employee can belong to just one department, the "department" relation is a to-one.

4. **Inverse relationship**: The definition of the inverse function. It is pretty rare to find a domain where this is not needed. It is also a sort of logical necessity: if an employee belongs to a department, it means that that department has employees. An inverse relation just switches the "direction" of the original relation.

   In your example, a department can have more than one employee, so this is a to-many relation. As a general rule, a one-to-many relation has a many-to-one inverse. In case you want to define a many-to-many relationship, you simply define one relation as to-many and its inverse as a to-many as well.

   Make sure you define an inverse for each relationship, since Core Data exploits this information to check the consistency of the object graph whenever a change is made.

5. **Delete rule**: This defines the behavior of the application when the source object of a relationship is deleted.

For the delete rule in #5 above, there are four possible values:

- **Nullify** is the simplest option. For example, if you delete a department, the "department" value of each employee previously belonging to that department is set to null. Nobody is fired :]
- **No action** means that, when you delete a department there is no change to the "department" value of each employee. They just keep thinking they have not been fired :]
- **Cascade** may have side effects, so you should use it carefully. If you select cascade as the delete rule, then when you delete the source object it also deletes the destination object(s). So, if you're shutting down a department but want to keep the employees, you should not use cascade. Such a rule is appropriate only if you want to close a department and fire all of its employees as well. In this case it is enough to set the delete rule for department to cascade and delete that department record.
- **Deny**, on the other hand, prevents accidental deletions. If you've set deny as the delete rule, before you can delete a department you have to make sure all its employee instances have been deleted or associated with another department.

Delete rules have to be specified for both sides of a relationship, from employee to department and vice versa. Each domain

implements its own business logic, so there is no general recipe for setting delete rules. Just remember to pay attention when using the cascade rule, since it could result in unexpected consequences.

For example, if both department -> employee (to-many) and employee -> department (to-one) are set to cascade, the deletion of a user triggers the deletion of a department which in turn fires back the deletion of all its employees! It is likely you don't want that. In this particular case, the deletion rule for employee -> department should be set to nullify.

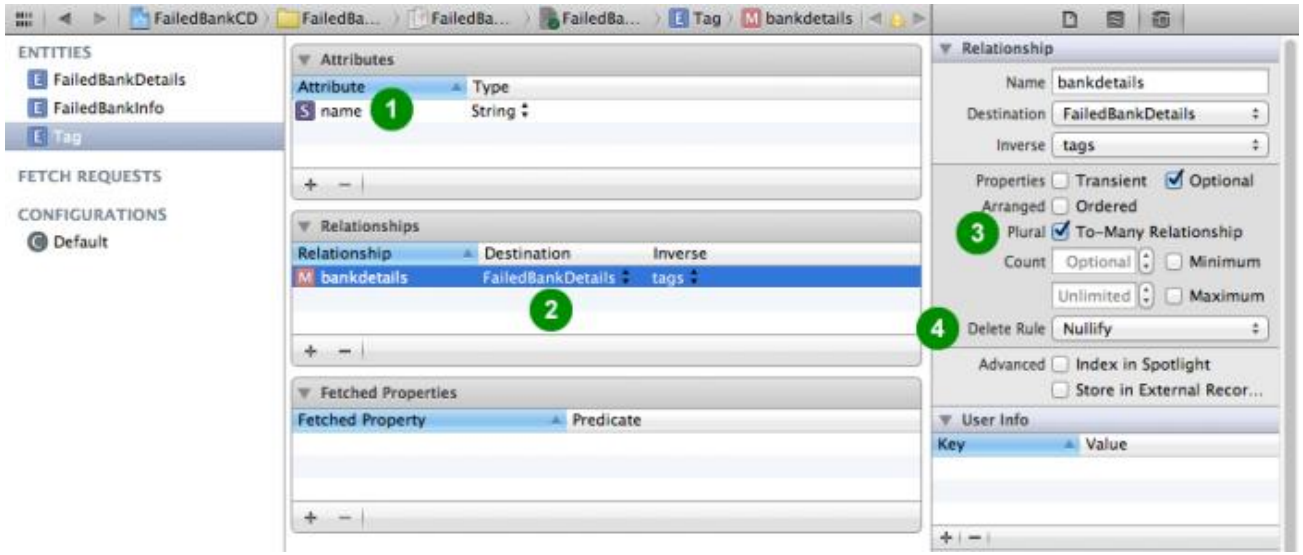
> **Note:** Even though you access a relationship via dot syntax, as if it were a property, it isn't: instead it corresponds to an actual query in the database. To maximize the performance of your application, remember this when you devise your data model and try to use relationships only if necessary.

Adding a Many-to-Many Relationship

Now you're going to extend your data model by adding a new entity, connected to the info object with a many-to-many relationship.

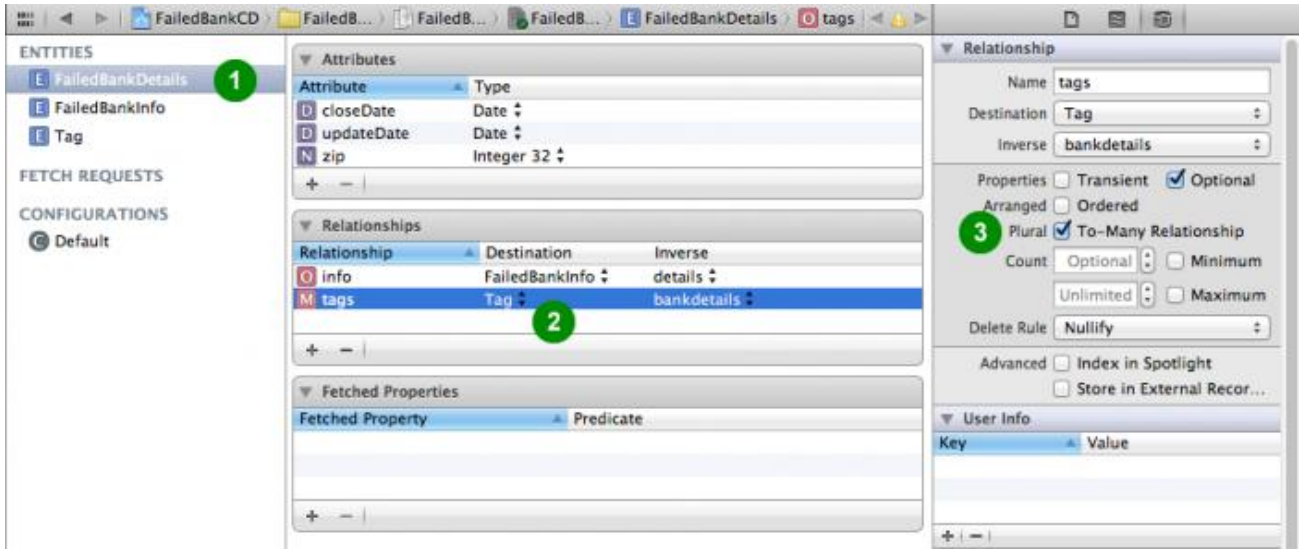
In a real-world scenario, you probably wouldn't model data this way. You're doing it in this tutorial only to cover a complex situation with predicates (see below). The first step is to add a new entity.

Open `FailedBankCD.xcdatamodeld` and add a new entity named "Tag." Then add a single attribute to it, "name" (1).

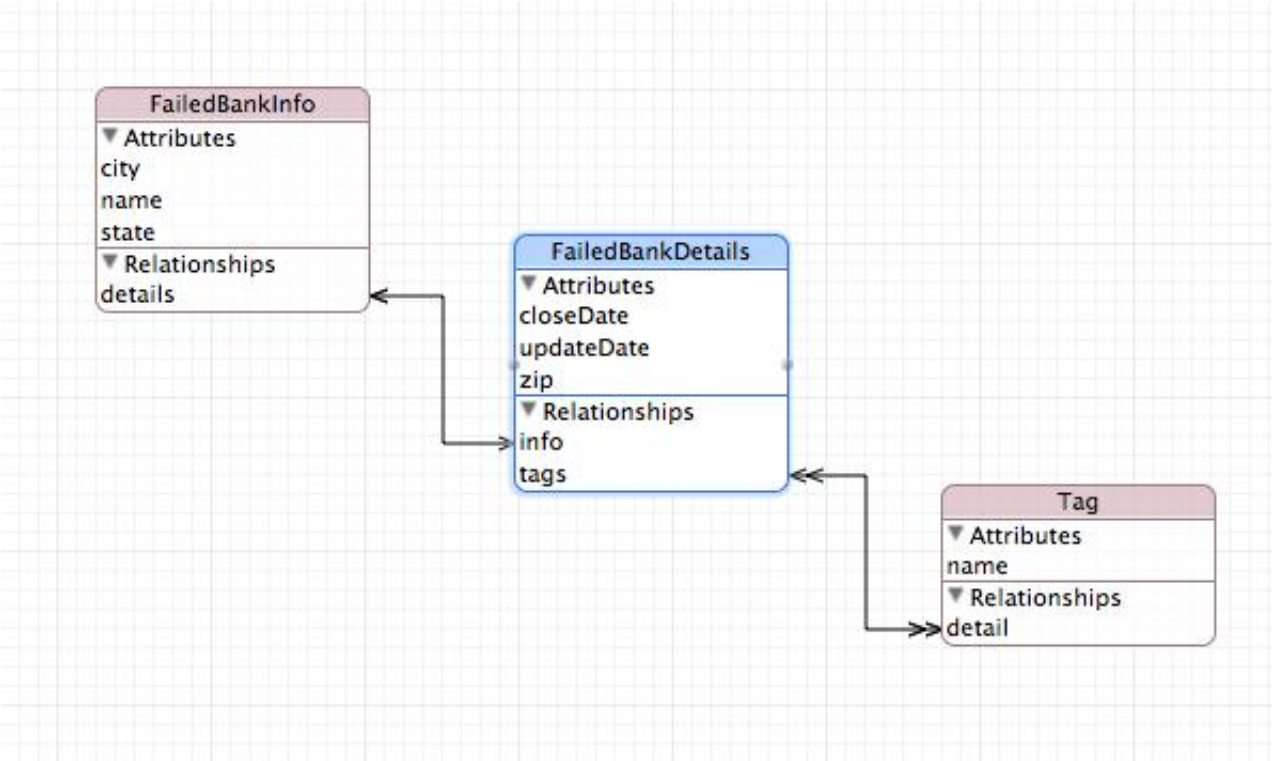


Then add a relationship, named "bankdetails," whose destination is FailedBankDetails (2) and set its type to to-many (3). The delete rule is the default, nullify (4). This means that if a tag is deleted, the "linked" detail objects are not deleted but simply lose a tag.

To define its inverse, select FailedBankDetails (1), add a new relationship called "tags" with Tag as the destination, and set the inverse to bankdetails (2). As above, this is a to-many relationship (3) with a delete rule of nullify.



You should end up with the following graphical model, where FailedBankDetails acts as a sort of bridge between FailedBankInfo and Tag.



With all three entities selected, open the Editor menu item and choose "Create NSManagedObject Subclass…" Then select your project folder as the save destination. Select "Replace" when asked to overwrite the definition of the previous classes. A new class, named Tag, will pop up in your project tree.

> **Note: Sometimes, (quite often, in fact :p) Xcode will mess up the code generation and add a second instance of existing Core Data entities to the project tree. If this happens to you, select one set of instances and delete them, but choose to remove references rather than to trash the files. There's actually only one set of physical files – so if you trash them, the other set of links might not work either.**

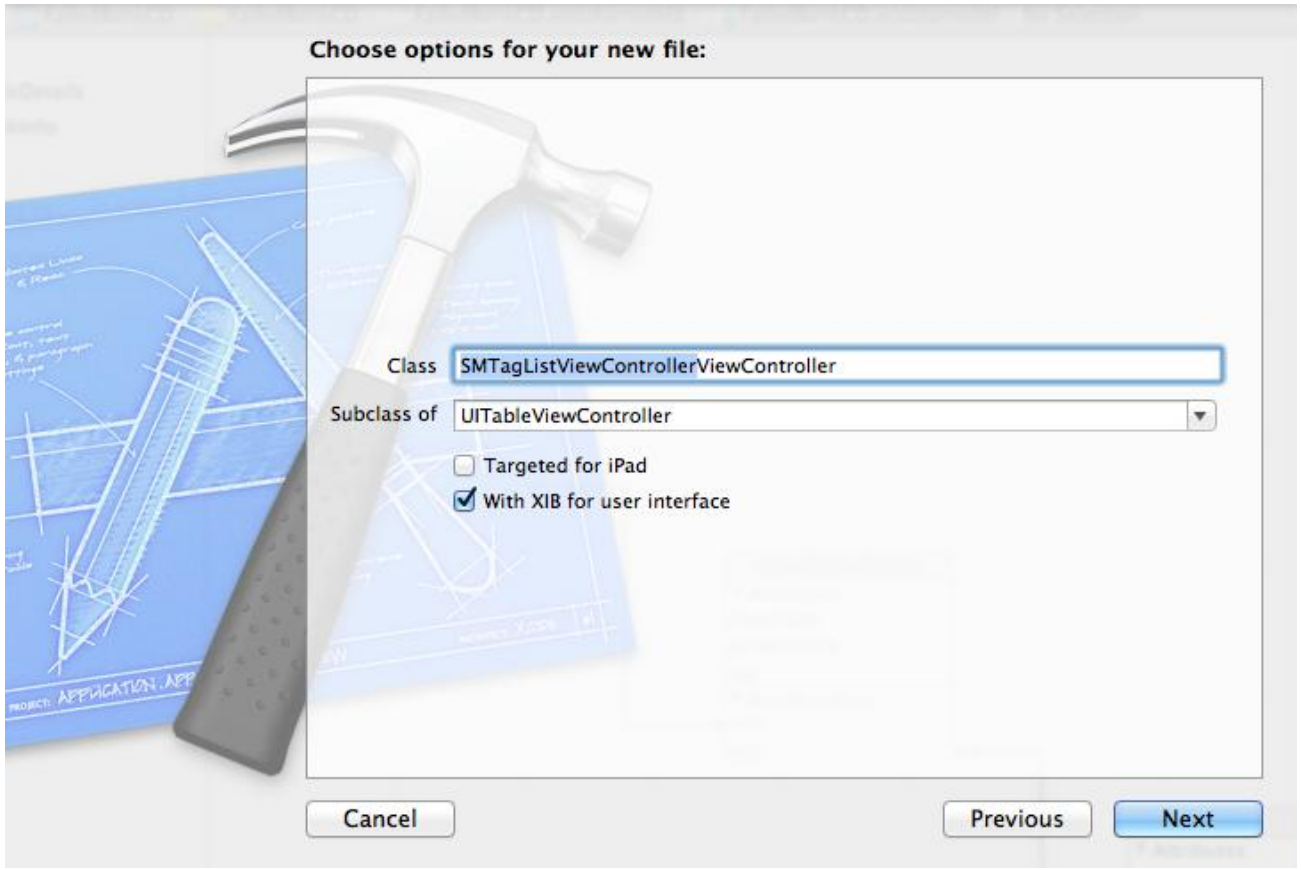
At this point, you have changed the Core Data model, so your app will not be compatible with the old model on your device.

There is a way around this with Core Data migrations, but that is a topic for another tutorial ;] For now, just delete the app off your device/simulator to get rid of any old files.

The next step is to build a view to create/edit tags associated with an instance of FailedBankDetails.

Tag, You're It!

This new view controller will facilitate the creation of new tags and associating them to a bank details object. Create a new class that extends UITableViewController and name it SMTagListViewController by right-clicking the root of the project and selecting New File…\iOS\Cocoa Touch\Objective-C class. Remember to check the box to create the accompanying XIB file.



Replace the contents of SMTagListViewController.h with the following:

```
#import <UIKit/UIKit.h>
#import "FailedBankDetails.h"
#import "Tag.h"

@interface SMTagListViewController : UITableViewController <UIAlertViewDelegate>

@property (nonatomic, strong) FailedBankDetails *bankDetails;
```

```
@property (nonatomic, strong) NSMutableSet *pickedTags;
@property (nonatomic, strong) NSFetchedResultsController *fetchedResultsController;

-(id)initWithBankDetails:(FailedBankDetails *)details;

@end
```

You import two needed classes, mark the view controller as implementing the UIAlertViewDelegate, and you add three properties: the bank details that refer to the previous screen, a set to collect the picked tags for the current details, and a results controller to fetch the whole list of tags. Finally, you add a method to initialize the component with an instance of details.

At the top of SMTagListViewController.m (below the @implementation line), synthesize the properties and implement initWithBankDetails:

```
@synthesize bankDetails = _bankDetails;
@synthesize pickedTags;
@synthesize fetchedResultsController = _fetchedResultsController;

-(id)initWithBankDetails:(FailedBankDetails *)details {
    if (self = [super init]) {
        _bankDetails = details;
    }
    return self;
}
```

The fetched results controller is defined to load all the instances of tags from the context. Add the code for it as follows to the end of the file (but before the final @end):

```
-(NSFetchedResultsController *)fetchedResultsController {
    if (_fetchedResultsController != nil) {
        return _fetchedResultsController;
    }
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];

    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Tag"
```

```
        inManagedObjectContext:self.bankDetails.managedObjectContext];
    [fetchRequest setEntity:entity];

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

    NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController alloc]
        initWithFetchRequest:fetchRequest managedObjectContext:self.bankDetails.managedObjectContext
        sectionNameKeyPath:nil cacheName:nil];
    self.fetchedResultsController = aFetchedResultsController;
        NSError *error = nil;
    if (![self.fetchedResultsController performFetch:&error]) {
            NSLog(@"Core data error %@, %@", error, [error userInfo]);
            abort();
        }

    return _fetchedResultsController;
}
```

The above is pretty similar to previously defined fetched results controllers – it's just for a different entity.

Replace the existing `viewDidLoad` with the following:

```
-(void)viewDidLoad {
    [super viewDidLoad];
    self.pickedTags = [[NSMutableSet alloc] init];
    // Retrieve all tags
    NSError *error;
    if (![self.fetchedResultsController performFetch:&error]) {
            NSLog(@"Error in tag retrieval %@, %@", error, [error userInfo]);
            abort();
        }
    // Each tag attached to the details is included in the array
    NSSet *tags = self.bankDetails.tags;
    for (Tag *tag in tags) {
```

```
        [pickedTags addObject:tag];
    }
    // setting up add button
    self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItem
        target:self action:@selector(addTag)];
}
```

Here you run a fetch operation and populate the set of pickedTags that are attached to the instance of bankDetails. You need this to show a tag as picked (by means of a tick) in the table view. You also set up a navigation item to add new tags.

Add the following below `viewDidLoad`:

```
-(void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    self.bankDetails.tags = pickedTags;
    NSError *error = nil;
    if (![self.bankDetails.managedObjectContext save:&error]) {
        NSLog(@"Error in saving tags %@, %@", error, [error userInfo]);
        abort();
    }
}
```

When the view is closed, you save the set of picked tags by setting the tags property of bankDetails.

Now add the following code to the end of the file:

```
-(void)addTag {
    UIAlertView *newTagAlert = [[UIAlertView alloc] initWithTitle:@"New tag"
        message:@"Insert new tag name" delegate:self cancelButtonTitle:@"Cancel" otherButtonTitles:@"Save", nil];
    newTagAlert.alertViewStyle = UIAlertViewStylePlainTextInput;
    [newTagAlert show];
}
```

To add a new tag, you use an alert view with an input text field. The code above will display an alert asking the user to insert a new tag:

To handle all actions for the alert view, add the following delegate method to the end of SMTagListViewController.m:

```
-(void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
    if (buttonIndex == 0) {
```

```
        NSLog(@"cancel");
    } else {
        NSString *tagName = [[alertView textFieldAtIndex:0] text];
        Tag *tag = [NSEntityDescription insertNewObjectForEntityForName:@"Tag"
            inManagedObjectContext:self.bankDetails.managedObjectContext];
        tag.name = tagName;
        NSError *error = nil;
        if (![tag.managedObjectContext save:&error]) {
            NSLog(@"Core data error %@, %@", error, [error userInfo]);
            abort();
        }
        [self.fetchedResultsController performFetch:&error];
        [self.tableView reloadData];
    }
}
```

You ignore a tap on the cancel button whereas you save the new tag if "OK" is tapped. In such a case, instead of implementing the change protocols to the table, you fetch the result again and reload the table view for the sake of simplicity.

Next replace the placeholders for `numberOfSectionsInTableView` and `tableView:numberOfRowsInSection` with the following:

```
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    id <NSFetchedResultsSectionInfo> sectionInfo = [[self.fetchedResultsController sections] objectAtIndex:section];
    return [sectionInfo numberOfObjects];
}
```

This is pretty straightforward – there is only one section, and the number of rows is calculated according to the results controller.

Next, modify `tableView:cellForRowAtIndexPath:` as follows:

```
-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"TagCell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
             reuseIdentifier:CellIdentifier];
    }
    cell.accessoryType = UITableViewCellAccessoryNone;
    Tag *tag = (Tag *)[self.fetchedResultsController objectAtIndexPath:indexPath];
    if ([pickedTags containsObject:tag]) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }
    cell.textLabel.text = tag.name;
    return cell;
    return cell;
}
```

This shows a checkmark if a tag belongs to the pickedTags set.

Finally, replace `tableView:didSelectRowAtIndexPath` with the following:

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    Tag *tag = (Tag *)[self.fetchedResultsController objectAtIndexPath:indexPath];
    UITableViewCell * cell = [self.tableView  cellForRowAtIndexPath:indexPath];
    [cell setSelected:NO animated:YES];
    if ([pickedTags containsObject:tag]) {
        [pickedTags removeObject:tag];
        cell.accessoryType = UITableViewCellAccessoryNone;
    } else {
        [pickedTags addObject:tag];
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }
}
```

This makes it so that when a cell is tapped, the corresponding tag is added to or removed from the set, and the cell updated accordingly.

Tagging Like a Fool

Take a deep breath – you're almost there! :]

Make the following modifications to SMBankDetailViewController.m:

```
// Add import at top of file
#import "SMTagListViewController.h"

// Add the following to the end of viewDidLoad
// 4 - setting interaction on tag label
self.tagsLabel.userInteractionEnabled = YES;
UITapGestureRecognizer *tagsTapRecognizer = [[UITapGestureRecognizer alloc] initWithTarget:self
    action:@selector(tagsTapped)];
[self.tagsLabel addGestureRecognizer:tagsTapRecognizer];
```

This adds a tap gesture recognizer so that you can get a callback when the user taps the tags label on the edit view.

Next implement this callback:

```
-(void)tagsTapped {
    SMTagListViewController *tagPicker = [[SMTagListViewController alloc] initWithBankDetails:self.bankInfo.details];
    [self.navigationController pushViewController:tagPicker
                                         animated:YES];
}
```

So when the tags label gets tapped, we present the SMTagListViewController we just made.

viewWillAppear: needs to be tweaked a bit to display tags correctly. At the bottom of the method implementation, add this code:

```
NSSet *tags = self.bankInfo.details.tags;
NSMutableArray *tagNamesArray = [[NSMutableArray alloc] initWithCapacity:tags.count];
for (Tag *tag in tags) {
    [tagNamesArray addObject:tag.name];
}
```

```
self.tagsLabel.text = [tagNamesArray componentsJoinedByString:@","];
```

This just makes a string of all of our tags separated by commas so we can display it.

As a final touch, make the label backgrounds gray to show their tappable area. Add this to the end of `viewDidLoad`:

```
    self.tagsLabel.backgroundColor = self.dateLabel.backgroundColor = [UIColor lightGrayColor];
```

You're done! Build and run your application and test it. Try the following:

1. **Add a new bank record.**
2. **Tap it.**
3. **Change its values.**
4. **Tap the tags label (will be empty the first time).**
5. **Add the tags you like.**
6. **Tap a few to associate them to the details object.**
7. **Tap the back button to verify that the details are correctly updated.**

> **Note: At this point, if haven't deleted the previous instance of your app as mentioned earlier, you might have a crash when you try to run it, with an error message saying, "The model used to open the store is incompatible with the one used to create the store."**
>
> **This happens because you changed the Core Data model since you last ran the app. You would need to delete the existing instance of the app on the simulator (or the device) and then compile and run your project. Everything should work fine at that point.**

Predicates: Having It Your Way

So far you have always fetched all the objects. Rather greedy, aren't you? :]

But what if you don't want everything? What if you want a subset, such as:

- All the banks whose names contain a given string.
- All the banks closed on a given date.
- All the banks whose zip codes end with a specific digit or string of digits.
- All the banks closed after a given date.
- All the banks with at least one tag.

These are just a few examples of the rich variety of queries that can be made to a database. And you can create even more complex queries by using AND/OR logical operators.

Well there's good news – you can easily do this in Core Data with something magical called a "predicate!"

A predicate is an operator whose job it is to return true or false. Whenever you have a list of objects that you want to filter, you can apply a predicate to the list.

This will apply the predicate condition (in other words, "filter criteria") to each one. It will return a subset (possibly empty) of the original list, with only the objects that matched the condition.

NSPredator... erm I mean NSPredicate!

In Objective-C, predicates are implemented via the NSPredicate class. There is a wide range of operators that can be used with NSPredicate. Operators are special keywords that allow defining a predicate. Each predicate has a format defined as a string.

The following, for example, defines a predicate that checks for the condition "has name equal to," where someName is a string variable containing the name to check for:

```
NSPredicate *pred = [NSPredicate predicateWithFormat:@"name == %@", someName];
```

The basic Objective-C code to use a predicate has the following format:

```
...
NSFetchRequest *fetchRequest = ... ;
NSPredicate *pred = ...;
[fetchRequest setPredicate:pred];
...
```

Here is a non-exhaustive list of predicate operators (a complete list is available ~~here~~):

- **CONTAINS: to query for strings that contain substrings.**
- **==: equality operator.**
- **BEGINSWITH: a pre-made regular expression that looks for matches at the beginning of a string.**
- **MATCHES: regular expression-like search.**
- **ENDSWITH: opposite of BEGINSWITH.**
- **<, >: less than and greater than.**

In the case of strings, it's also possible to specify case-sensitivity. By default BEGINSWITH and the like are case sensitive. If you are not interested in the case, you can use the [c] key to specify a case-insensitive search. For example, the following looks for a string beginning with the value contained in "someName," regardless of the case:

```
pred = [NSPredicate predicateWithFormat:@"name BEGINSWITH[c] %@", someName];
```

So if "someName" contained the value "cat," it would match both "catatonic" and "catacombs." Wow, those are rather dark words! I suppose it would also match words like "caterpillar" and "catnap," for those of you with sunnier dispositions. :]

Integrating Predicates With the App

Now you're going to build a new view controller that lets the user run searches on the database of banks. Create a new file

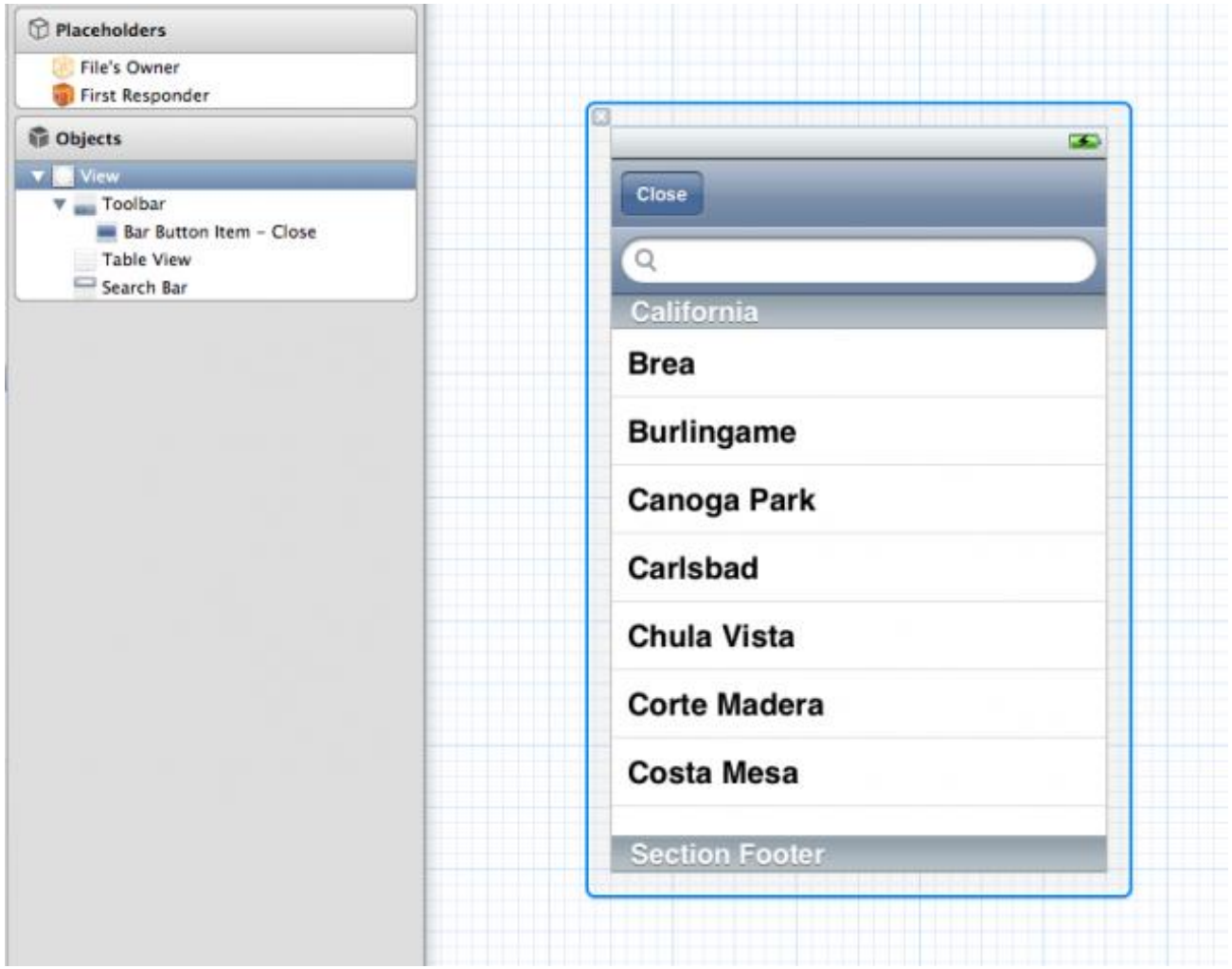
using the Objective-C class template. This new class will be called SMSearchViewController and will extend UIViewController. And remember to create a XIB file to match the class.

Replace the contents of SMSearchViewController.h with the following:

```
#import <UIKit/UIKit.h>
#import "FailedBankInfo.h"

@interface SMSearchViewController : UIViewController<UITableViewDelegate, UITableViewDataSource,NSFetchedResultsControl

@property (nonatomic,strong) NSManagedObjectContext* managedObjectContext;
@property (nonatomic,retain) NSFetchedResultsController *fetchedResultsController;
@property (nonatomic, strong) IBOutlet UISearchBar *searchBar;
@property (nonatomic, strong) IBOutlet UITableView *tView;
@property (nonatomic, strong) UILabel *noResultsLabel;

-(IBAction)closeSearch;

@end
```

Here you give the view controller references to a context to run the searches, a search bar, a table view and their respective protocols.

Switch to SMSearchViewController.xib and add a toolbar, a table view and a search bar, and link them to the respective outlets you defined above. The final screen should look something like this:



Switch to SMSearchViewController.m to synthesize the properties and define a helper method you'll implement later:

```
@interface SMSearchViewControllerViewController ()
-(void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath;
```

```
@end

@implementation SMSearchViewControllerViewController

@synthesize managedObjectContext;
@synthesize fetchedResultsController = _fetchedResultsController;
@synthesize searchBar,tView;
@synthesize noResultsLabel;
```

Add the code for `closeSearch`, which simply dismisses the view controller, to the end of the file:

```
-(IBAction)closeSearch {
    [self dismissModalViewControllerAnimated:YES];
}
```

In `viewDidLoad`, assign the delegate to the table and the search bar, and initialize the noResultsLabel:

```
-(void)viewDidLoad {
    [super viewDidLoad];
    self.searchBar.delegate = self;
    self.tView.delegate = self;
    self.tView.dataSource = self;

    noResultsLabel = [[UILabel alloc] initWithFrame:CGRectMake(20, 90, 200, 30)];
    [self.view addSubview:noResultsLabel];
    noResultsLabel.text = @"No Results";
    [noResultsLabel setHidden:YES];
}
```

When the view appears, display the keyboard:

```
-(void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [self.searchBar becomeFirstResponder];
}
```

Once the user taps "Search" on the keyboard (after typing in a search value, of course), you run a fetch and show the results on the table view, or display the "No results" label. Add the following method to the end of the file to do that:

```
-(void)searchBarSearchButtonClicked:(UISearchBar *)searchBar {
    NSError *error;
        if (![[self fetchedResultsController] performFetch:&error]) {
                NSLog(@"Error in search %@, %@", error, [error userInfo]);
        } else {
        [self.tView reloadData];
        [self.searchBar resignFirstResponder];
        [noResultsLabel setHidden:_fetchedResultsController.fetchedObjects.count > 0];
    }
}
```

The table view dataSource methods are pretty intuitive and fairly routine. You just display the cell as in the master view controller.

```
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    id  sectionInfo =
    [[_fetchedResultsController sections] objectAtIndex:section];
    return [sectionInfo numberOfObjects];

}

-(void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath {
    FailedBankInfo *info = [_fetchedResultsController objectAtIndexPath:indexPath];
    cell.textLabel.text = info.name;
    cell.detailTextLabel.text = [NSString stringWithFormat:@"%@, %@",
                                 info.city, info.state];
}

-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (!cell) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier:CellIdentifier];
    }
    [self configureCell:cell atIndexPath:indexPath];
    return cell;
}
```

Now you're left with the core functionality for the view: the fetched results controller with a predicate. Add the following code to the end of the file:

```
-(NSFetchedResultsController *)fetchedResultsController {
    // Create fetch request
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"FailedBankInfo"
        inManagedObjectContext:managedObjectContext];
    [fetchRequest setEntity:entity];
    NSSortDescriptor *sort = [[NSSortDescriptor alloc] initWithKey:@"details.closeDate" ascending:NO];
    [fetchRequest setSortDescriptors:[NSArray arrayWithObject:sort]];
    [fetchRequest setFetchBatchSize:20];
    // Create predicate
    NSPredicate *pred = [NSPredicate predicateWithFormat:@"name CONTAINS %@", self.searchBar.text];
    [fetchRequest setPredicate:pred];
    // Create fetched results controller
    NSFetchedResultsController *theFetchedResultsController = [[NSFetchedResultsController alloc] initWithFetchRequest:
        managedObjectContext:managedObjectContext sectionNameKeyPath:nil cacheName:nil]; // better to not use cache
    self.fetchedResultsController = theFetchedResultsController;
    _fetchedResultsController.delegate = self;
    return _fetchedResultsController;
}
```

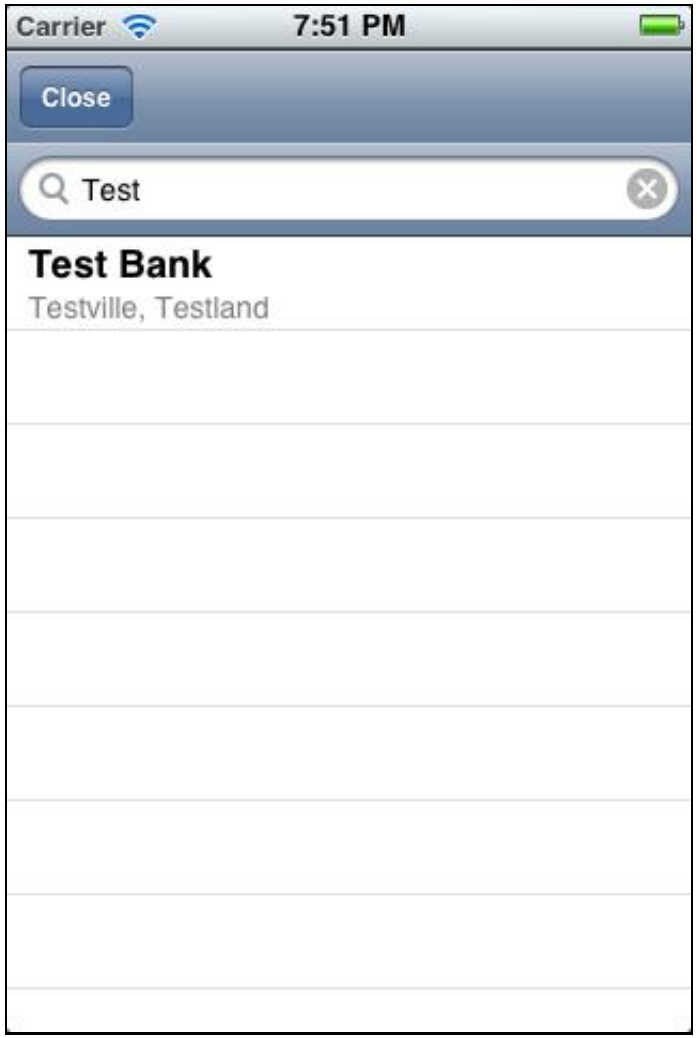
The first part is pretty similar to what you've already seen: you create a request, specify an entity, and assign a batch size to it. Then you get to choose which predicate to play with. The code above uses CONTAINS. You assign the predicate to the fetch request and then create and return a fetched results controller, as usual.

The final step is, of course, to implement the search functionality in the main view :] Switch to FBCDMasterViewController.m and add the following code:

```
// Add at the top of the file under the imports section
#import "SMSearchViewController.h"

// Add at the bottom of the file before @end
-(void)showSearch {
    SMSearchViewController *searchViewController = [[SMSearchViewController alloc] init];
    searchViewController.managedObjectContext = managedObjectContext;
    [self.navigationController presentModalViewController:searchViewController
                                                 animated:YES];
}
```

Time to test your code again!

Compile and run the application, and look for banks whose names start with the string typed into the search bar. Remember that by default the CONTAINS operator is case-sensitive. Here's an example of this version of the app in action:



Pretty powerful stuff!

More Fun with Predicates

If you want to search for a name that matches the search term exactly, then modify the predicate creation line as follows:

```
NSPredicate *pred = [NSPredicate predicateWithFormat:@"name == %@", self.searchBar.text];
```

Want to try something a bit more complicated? Remember that your data objects aren't alone: they have relationships. You can define a predicate referring to a value accessed via a relationship. For example, the following looks for zip codes ending with whatever the user types into the search box.

```
NSPredicate *pred = [NSPredicate predicateWithFormat: @"details.zip ENDSWITH %@", self.searchBar.text];
```

Note that you can access relationships using dot notation – much like Objective-C properties. In this example, you're checking the zip property of the bank details.

Not complicated enough? How about checking for banks closed after a given date? Maybe you want to see how many have closed since the beginning of the year.

```
NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
[dateFormatter setDateFormat:@"yyyy-MM-dd"];
NSDate *date = [dateFormatter dateFromString:self.searchBar.text];
NSPredicate *pred = [NSPredicate predicateWithFormat: @"details.closeDate > %@", date];
```

Here you build a date from the input string, and use the > operator.

You can even exploit a "chain" of objects going down to the tag level. For example, if you want to retrieve banks that have more than one tag, you can use the @count operator, which works on properties modeled as a set.

```
NSPredicate *pred = [NSPredicate predicateWithFormat: @"details.tags.@count > 0"];
```

As mentioned before, you can use AND/OR logical operators to combine filter criteria. Let's assume the user will provide two values split by a colon (:). For example, to look for a bank by name and city, he types in "bank:testville". That can be handled as follows:

```
    NSArray *queryArray;
    if ([self.searchBar.text rangeOfString:@":"].location != NSNotFound) {
        queryArray = [self.searchBar.text componentsSeparatedByString:@":"];
```

```
    }
    NSPredicate *pred = [NSPredicate predicateWithFormat:@"(name CONTAINS[c] %@) AND (city CONTAINS[c] %@)",
        [queryArray objectAtIndex:0], [queryArray objectAtIndex:1]];
```

First you create an array with the two values by splitting the search string into two. Then you build a predicate using the two separate values and the AND logical operator. The above predicate can be translated as: "look for the banks whose names contain the string x and whose city names contain the string y."

The final source code archive for this tutorial contains a put a constant called SEARCH_TYPE, which ranges from 1 to 11 and allows you to experiment with different operators for predicates. Check it out in the download link below!

Where To Go From Here?

Here is a ~~sample project~~ with all of the code from this tutorial so far.

I hope you're feeling more like a wizard when it comes to relations and predicates.

Want to experiment some more? Instead of "chaining" predicates in a string, you might want to explore the convenient classes NSCompoundPredicate and NSComparisonPredicate. These allow you to achieve the same results in code.

If you have any questions or comments about this tutorial or relationships/predicates in general, please join the forum discussion below!

This is a post by iOS Tutorial Team member ~~Cesare Rocchi~~, a UX designer and developer specializing in web and mobile applications.

iPhoneCategory:

Tags: Core Data, iOS, iPhone, sample code, tutorial

I'd love to hear your thoughts!

~~13 Comments!~~ **~~Add a comment!~~**