Extracted from:

iOS SDK Development

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

iOS SDK Development

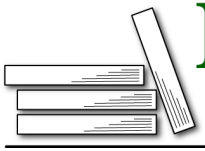
Chris Adamson
and Bill Dudney

Edited by Brian P. Hogan

iOS SDK Development

Chris Adamson
Bill Dudney

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-94-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2012

5.2 Displaying a List of Recipes

We left our recipes application in the last chapter displaying a single recipe. While that was a great start, what we really want is to keep a whole recipe box on our iPhone. To pull that off, we are going to need several instances of our PRPRecipe model. We will then need a view that is capable of displaying a list of objects, and for that, we'll use a table view. As before, we'll use MVC: if the table is the view and we have a model that holds several recipes, then we need a view controller to combine the two. Let's get started by creating the controller and the view, and then we will create our list of model objects.

Create a Table View Controller

First we need to create a subclass of UITableViewController to translate between our list of recipes and the table view that displays them. UITableViewController is a subclass of UIViewController that defines additional methods that support a table view.

In Xcode with the PRPRecipes project open, let's select the "File->New File..." menu item (⌘N). In the sheet that drops down, we need to choose the "Objective-C class" item and then click Next. Since we are creating a table view controller, we need to choose UITableViewController in the "Subclass of" pull-down. Let's call the new class PRPRecipesListViewController. Be sure that the "With XIB for user interface" check box is also selected. Click Next, and in the next page, choose the Recipes folder and then click the Create button.

You should see three new files in Xcode's project navigator: the PRPRecipesListViewController.xib file and the PRPRecipesListViewController header and implementation files (.h and .m).

Create Recipes List

In the previous version of this example, we created a single instance of PRPRecipe to serve as the model, which the controller then used to populate the view. That worked great then, but now we have a list to fill. We could use the one recipe as a table model, but that single item would be lonely. Let's go back to the PRPAppDelegate and modify it to make and keep track of several recipes.

First we need to create a property on the app delegate to hold the list. The natural fit for a list of objects is an NSArray. In Xcode, select the PRPAppDelegate.h file and add a property for the new list, like this:

TableViews/Recipes_04/Recipes/PRPAppDelegate.h

```
@property (copy, nonatomic) NSArray *recipes;
```

Lazy Creation

Let's create the list and then place it into the recipes property. We could do that in the app delegate and assign the view controller's property, or we could create it in the VC's init method. But let's try something different, something lazy.

Often, it's a good idea to create things "lazily," meaning only when we absolutely need them. When we create objects in our program at the last minute, we keep from allocating the memory and doing the work to initialize them until the last possible minute. This typically works in our favor because we don't use memory until we have to. Lazy creation also makes it easier to clean it up when we are done because we know if we delete the objects they will be recreated when we need them.

To implement our recipes method that lazily creates the array, we go to PRPAppDelegate.m and write a new getter method, like this:

TableViews/Recipes_05/Recipes/PRPAppDelegate.m

```
- (NSArray *)recipes {
  if(!_recipes) {
    // create the recipes here
  }
  return _recipes;
}
```

Remember that properties in Objective-C are just declarations that specify the way the get and set methods work. So what we have done here is to provide the get method rather than let the compiler generate it for us (the setter will still be synthesized). The first thing we do here is to check to see if the instance variable (also called an *ivar* for short) backing the property exists. If the ivar exists, then the _recipes array exists and we can simply return it.

If the ivar doesn't exist, we need to create an array, fill it with recipes, and then assign our recipes property to the newly created array. Import the PRPRecipe.h filename and then replace the comment with this code to create the array and add the first two recipes:

TableViews/Recipes_04/Recipes/PRPAppDelegate.m

```
NSMutableArray *localRecipes = [NSMutableArray array];
PRPRecipe *recipe = [[PRPRecipe alloc] init];
recipe.directions = @"0 - Put some stuff in, and the other stuff, then stir";
recipe.title = @"0 - One Fine Food";
recipe.image = [UIImage imageNamed:@"cookies.jpg"];
[localRecipes addObject:recipe];
recipe = [[PRPRecipe alloc] init];
recipe.directions = @"1 - Put some stuff in, and the other stuff, then stir";
recipe.title = @"1 - One Fine Food";
```

```
recipe.image = [UIImage imageNamed:@"cookies.jpg"];
[localRecipes addObject:recipe];
```

The code download bundle creates several recipes, as shown in the following screenshots. You are, of course, free to create five or twenty-five recipes in this method; all of them will show up in your tableview.

The final step in our lazy creation code is to assign our localRecipes variable to the recipes property. After the last recipe has been added to localRecipes, add this line of code.

TableViews/Recipes_04/Recipes/PRPAppDelegate.m

```
self.recipes = localRecipes;
```

Since the recipes property is specified to copy its value, writing self.recipes = localRecipes will make a *copy* of localRecipes and assign it to the _recipes ivar. Making a copy of an NSMutableArray might sound inefficient, but it really isn't. The recipes are not copied, just the data structure to hold onto the list. It is important to make a copy, though, so that we end up with an immutable NSArray rather than its mutable subclass, since some other part of the app could change a mutable array without the view controller knowing about it.

We can add as many recipe instances here as we'd like, and it could be interesting to experiment with the number of recipes and see how the table view reacts. Speaking of the table view, let's go fix up the application:didFinishLaunchingWithOptions: method to display our new table view instead of the old single recipe view.

Display the Table View

Let's navigate to the application:didFinishLaunchingWithOptions: method in the PRPAppDelegate and change it so that our table view controller is the rootViewController. That will make the table the view that gets shown when the app comes up. Here's the updated implementation:

TableViews/Recipes_04/Recipes/PRPAppDelegate.m

```
Line 1 - (BOOL)application:(UIApplication *)application
  2 didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
  3   self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
  4   self.viewController = [[PRPRecipesListViewController alloc]
  5                          initWithNibName:@"PRPRecipesListViewController"
  6                          bundle:nil];
  7   self.window.rootViewController = self.viewController;
```

The big change here is on lines 4–6. If we were to build and compile now, two undesired things would happen. First, we'd get a compile error that our table view controller is undefined. Second, the compiler would complain about the

type mismatch between the viewController property and the type of the object we are assigning. To fix the first issue we need to change the #import statement to import the PRPRecipesListViewController.h instead of the PRPViewController.h in the header file. Next we need to modify the property declaration so the viewController's type matches. When done, the changed header should look like this:

TableViews/Recipes_05/Recipes/PRPAppDelegate.h

```
@class PRPRecipesListViewController;
@interface PRPAppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) PRPRecipesListViewController *viewController;
@property (copy, nonatomic) NSArray *recipes;
@end
```

If we were to build and run the app now, the table view would display, but it would not have any data in it. That's no good! We need to pass the array of recipes off to the table view controller. However, the view controller does not have a way for us to tell it what the recipes are. Right after the line to change the rootViewController, add a line of code like this:

TableViews/Recipes_04/Recipes/PRPAppDelegate.m

```
self.viewController.recipes = self.recipes;
```

Of course this won't compile, since the viewController does not have the recipes property defined on it yet. That's an easy fix though. First, add the property to the PRPRecipesListViewController. Open the header file and add the property declaration @property (strong, nonatomic) NSArray *recipes.

Now our table view controller knows about the list of recipes, but we still need to have it use that list to populate the table view.

A table view relies on an object that implements the UITableViewDataSource protocol to get its data. Fortunately, our PRPRecipesListViewController already implements this protocol as part of the template code for a subclass of UITableViewController. Let's go take a look and change the implementation to work with our new recipes property.

To update the code, we need to edit the implementation file, so open PRPRecipesListViewController.m in Xcode. We've been ignoring it up to this point, but if we were to hit the Run button in Xcode since adding this new controller, we would see two new build warnings. Both of them are coming from this class, warning us that two of the required methods still have the template-provided implementation. The first is numberOfSectionsInTableView:. This method is called by the table view to discover how many sections it has. We will cover sections in Chapter 6, *Storyboards and Container Controllers*, on page ?. For now, just change this method to return a constant value of 1, like this:

TableViews/Recipes_04/Recipes/PRPRecipesListViewController.m

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

The next warning is coming from the tableView:numberOfRowsInSection: method. This method is called by the table view for each section to determine how many rows will be in that particular section. Since our previous method implementation tells the table there is only one section, the section parameter sent to this method will always be 0. And for that one section, we return the number of rows as the length of the recipes array. Change your code to look like this:

TableViews/Recipes_04/Recipes/PRPRecipesListViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section {
    return [self.recipes count];
}
```

Now our table view knows how many rows it needs to display. The next step is to configure each row to display the correct data. The table view calls the tableView:cellForRowAtIndexPath: method of its data source for every row on the screen. The table view expects this method to return a UITableViewCell (or a subclass) for each row.

The table view is careful to only call this method for the rows that are visible onscreen. So if we had one thousand recipes, the table view would only call this method about ten times, once for each cell that is visible. Then as we scrolled through the list of a thousand recipes, it would ask us to provide another cell each time a row becomes visible.

Now that we know the conditions under which this method is called, all we need to do is update it to set the text to display in the cell. Change your code to look like this:

TableViews/Recipes_04/Recipes/PRPRecipesListViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView
                           dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                  reuseIdentifier:CellIdentifier];
  }

  cell.textLabel.text = [[self.recipes objectAtIndex:indexPath.row] title];
```

```
-   return cell;
- }
```

We only have to add one line of code to this method to make our list of recipes show up. We get the recipe we are displaying by grabbing the row from the indexPath passed into this method. Instances of NSIndexPath are used to identify a particular cell by its section and row properties. Then, to set the text of a cell, change the text property of the cell's textLabel, as on line 12.

Considering we only have to populate one property, the code provided for this method by the template seems to be doing a lot of extra work. To understand that, we need to look at how UIKit manages cell reuse.

Let's look back at line 3. Here we are declaring a string called CellIdentifier, and on line 6 we use that identifier to ask the table view for a reusable cell. Behind the scenes the table view is keeping track of all the cells that have been returned from tableView:cellForRowAtIndexPath:. When one of them scrolls offscreen, it gets pulled out of the table view and placed into the reuse queue. Then, as our controller is asked to provide cells to the table view via the tableView:cellForRowAtIndexPath: method, it asks the table view if there are any reusable cells available. If there is a suitable cell, one whose identifier matches, the table view returns that cell and our method just sets the cell label text. If there is not a cell to return, then we get back nil. In that case, we go to line 9, where we use the same identifier to create a new cell.

Cell reuse allows our app to only have a few cells in memory at a time; more importantly, it keeps us from having to allocate new memory for each cell during scrolling. Reusing cells is the only way we can keep our scrolling animations smooth.

And that's it—we now have our table view and its data source set up to display our list of recipes. Hit Run in Xcode and you should see something that looks like Figure 41, *First table view*, on page 11.

In this section we took the first step into the deep waters of the UITableView, but there is so much left to learn! Let's get started on our next step to table view mastery by looking at how to edit a table view.

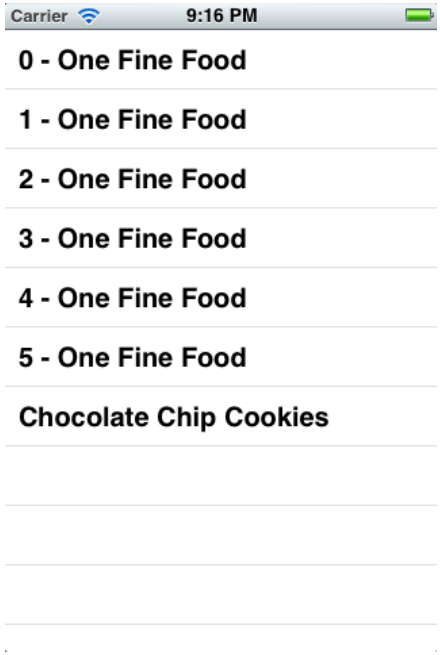


Figure 41—First table view