

Extracted from:

iOS SDK Development

This PDF file contains pages extracted from *iOS SDK Development*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

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



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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>.

Copyright © 2011 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-13: 978-1-934356-94-4

Printed on acid-free paper.

Book version: B1.0—December 14, 2011

4.3 Detailed Recipe

Now that we know how our view gets on screen, let's flesh out our recipe model a bit so that we have something nicer to look at. Along the way, we'll learn a bit about some of the other subclasses of `UIView` in `UIKit`.

Open the `PRPRecipe.h` header and add a property for the directions and a property for an image of the completed recipe. The code should look like this:

```
Download ViewsAndControllers/Recipes_03/Recipes/PRPRecipe.h
```

```
@interface PRPRecipe : NSObject

@property(nonatomic, copy) NSString *title;
@property(nonatomic, copy) NSString *directions;
@property(nonatomic, retain) UIImage *image;

@end
```

Be sure to use `retain` rather than `copy` on the `UIImage` property; it would waste memory to copy a lot of image data around. Also don't forget to add the `@synthesize` statements for both new properties to the implementation of the `PRPRecipe` class.

With these two properties in place, we can step up our game for the look of our view. Open the `RecipesViewController.xib` file by selecting it in the project navigator. We want to do three things here: first make the title use a bold font, add a text view so we can display the directions, and put an image view on the screen so we can show what the completed recipe looks like.

To change the title to use a bold font, we just need to change the font that the `UILabel` uses. Select the title label, open the Attributes Inspector (`⌘4`), and click on the "Change Font" button, as seen in [Figure 29, Changing Font of a UILabel, on page 6](#):

When the pop-up window is visible select the Font pop-up menu and then select the "System Bold" item.

Next, let's add a text view to display the directions. To do that, we need to go to the Object Library (`⌘3`). In the filter field at the bottom of this pane, type "text" to narrow the displayed objects to just the ones with "text" in their name or description.

Click on the "Text View" object and drag it onto the view. Using the Size Inspector (`⌘5`), place it at `{20, 50, 280, 100}`. In the Attributes Inspector, (`⌘4`) turn off editing, turn off horizontal scrollers, make it un-editable, and change the background color to clear.

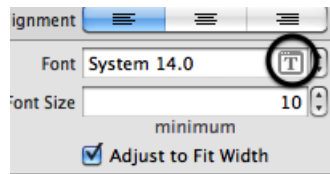


Figure 29—Changing Font of a UILabel

Next we want to place an image view. In the search field, type “image”, select the Image View, and drag it into the view. Size the view with the Size Inspector at {20, 160, 280, 280}. The interface should look like [Figure 30, Updated Interface, on page 7](#)

Now that we have some new views on the screen to show additional data, we need to wire up the controller to get the data from the model into these views. That means it’s time to head back over to PRPRecipesViewController.h. Select the file in Xcode and add two new IBOutlet properties, like this:

```
Download ViewsAndControllers/Recipes_03/Recipes/PRPViewController.h
```

```
@property(n nonatomic, retain) IBOutlet UITextView *directionsView;
@property(n nonatomic, retain) IBOutlet UIImageView *imageView;
```

Don’t forget to put corresponding @synthesize statements in the implementation file for these two new properties.

Now that the view controller has outlets for these two elements of the view, we need to head back to the interface file to make the connections to the text view and image view that we added just a couple of paragraphs back. Open PRPRecipesViewController.xib again, select the File’s Owner, and control-click and drag to the text view. Let up on the mouse and select directionsView from the popup. Do the same for the image view, but connect it to the imageView outlet. The Connections Inspector should look like [Figure 31, Updated Connections, on page 7](#)

Now we move on to updating the code in our controller to put the data we get from the model into the view. Open the PRPRecipesViewController.m and go back to the viewWillAppear: method. We need to grab the directions text from the recipe and place that into the directionsView, and grab the image of the completed recipe and place that into the imageView. The code is very simple; just rewrite viewWillAppear: like this:

```
Download ViewsAndControllers/Recipes_03/Recipes/PRPViewController.m
```

```
Line 1 - (void)viewWillAppear:(BOOL)animated {
2     [super viewWillAppear:animated];
3     self.recipeTitle.text = self.recipe.title;
```



Figure 30—Updated Interface

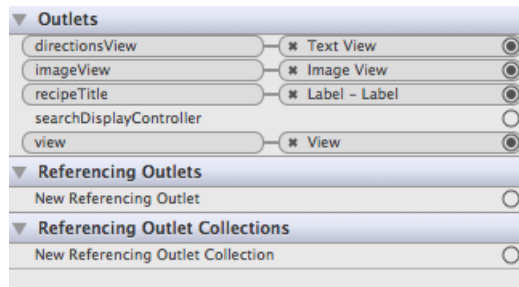


Figure 31—Updated Connections

```

4 self.directionsView.text = self.recipe.directions;
5 self.imageView.image = self.recipe.image;
6 }

```

This code is not much different from our previous version of `viewWillAppear:`. On line 4 we put the directions string from the recipe into the text property of the `directionsView` which simply places that string into the text view.

Next on line 5 we put the image from the recipe into the `imageView`'s image property.

We are done with the heavy lifting, but we still need to put an image and directions into the recipe model object. Remember that we created an instance of `PRPRecipe` in the `PRPRecipesAppDelegate`'s `application:didFinishLaunchingWithOptions:` method. So let's go back there and update that code. Open the file `PRPRecipesAppDelegate.m` and select the `application:didFinishLaunchingWithOptions:` method.

The downloadable sample code has an image and some directions for chocolate chip cookies. Feel free to do something similar or just copy what is in the sample code. To add an image file to Xcode, you can drag it from the Finder to the Project Navigator. When prompted, make sure to check the "Copy items into destination group's folder (if needed)" checkbox; this keeps the image file with your source files, rather than just managing a reference to the original image on your file-system. Don't forget to resize and crop the image to be 280x280.

The updated code should look like this:

Download ViewsAndControllers/Recipes_03/Recipes/PRPAppDelegate.m

```
Line 1 - (BOOL)application:(UIApplication *)application
- didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
-   NSString *directions = @"Put the flour and other dry ingredients in a bowl, \
-   stir in the eggs until evenly moist. Add chocolate chips and stir in until even. \
5   Place tablespoon sized portions on greased cookie sheet and bake at 350° for \
-   6 minutes.";
-   PRPRecipe *recipe = [[PRPRecipe alloc] init];
-   recipe.title = @"Chocolate Chip Cookies";
-   recipe.image = [UIImage imageNamed:@"IMG_1948.jpg"];
10  recipe.directions = directions;
-   self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds];
-   self.viewController = [[PRPViewController alloc]
-                           initWithNibName:@"PRPViewController"
-                           bundle:nil];
15  self.viewController.recipe = recipe;
-   self.window.rootViewController = self.viewController;
-   [self.window makeKeyAndVisible];
-   return YES;
- }
```

Sizing Images

Images are a big part of making iOS application stand out but they can be a huge performance drain. An often overlooked performance issue is the expense of compressing an image. In the sample code the image of the chocolate chip cookies is 280x280. The image was purposefully sized that way to match the size of the UIImageView instance in our view. If we were to try and display a 5 mega-pixel image in a 280x280 image view, it would display just the same but the image view would have to do a bunch of work to make the image fit into that 280x280 size. In essence it would have to read 5 million points from the image to fill 78K points on the screen. That is very wasteful, both in terms of performance (how long it takes) as well as power consumption (how much power is drained from the battery). On the iPhone 4 you might not notice the performance hit because it's so fast and we are only displaying one image, but our users would notice the battery drain.

The directions string is set up on line 6. The “\” character tells the compiler that the string continues on the next line.¹ On line 9, we create a UIImageView from the IMG_1948.jpg image. Feel free to use this image, or take a break from coding and go bake your own cookies and take a picture of them.

Now our app is complete, go ahead and click the Run button in Xcode. The running app should look like [Figure 32, Completed Recipes App, on page 10](#).

Congratulations again! You have a view controller that is capable of showing a recipe including a photo of what the completed product will look like. Before we go onto the next topic we need to take a short detour into understanding how view controllers help us to manage the memory used by its view.

Memory Management

In the example thus far, we have not had to think much about memory management. With only one view controller, one model object and a relatively simple view, we just don't have enough going on yet to be concerned. But forming good habits up front is the key to keeping those good habits in the trenches.

As we said earlier, when iOS determines that a low memory situation is unfolding, it will send out a notification which all view controllers get in the form of the `didReceiveMemoryWarning` method call. This is a great override point to clean up any cached state that isn't currently needed. It's important to implement this method and clean up what can be cleaned up. Granted, in our simple

1. We use the backslash character to suit the formatting of the book; you can keep the string all on one line in your own code.

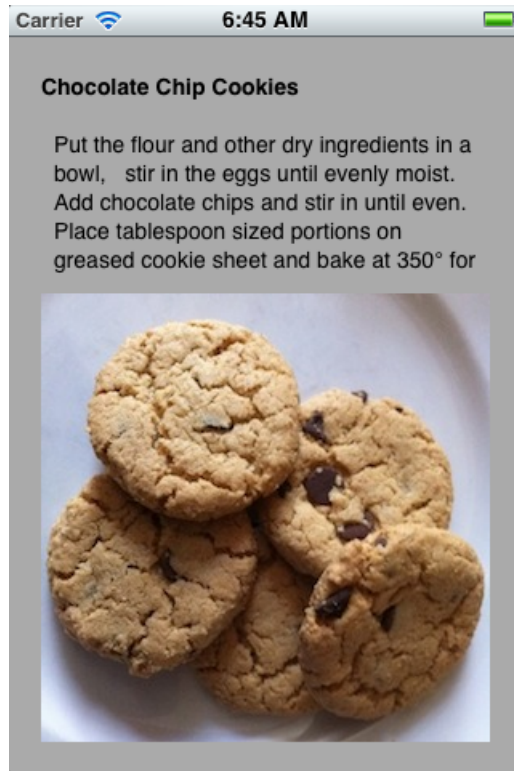


Figure 32—Completed Recipes App

example, it's hard to imagine a situation where this view controller could clean up anything. When it matters is for more complex view controllers that have a lot more state, or could afford to jettison their models and rebuild them later.

There are also memory-management considerations in the loading and unloading of views. A view controller will automatically unload its view if a memory notification is posted and its view is not on screen. After the view is unloaded and released, the view controller's `viewDidUnload` method is called. We should almost always override this method and release all objects that we created in `viewDidLoad` as well as any `IBOutlet`s that we have connections to on the view. Change the code to look like this:

Download ViewsAndControllers/Recipes_03/Recipes/PRPViewController.m

```
- (void)viewDidUnload {
    [super viewDidUnload];
    self.directionsView = nil;
    self.recipeTitle = nil;
}
```

```
    self.imageView = nil;  
}
```

First we make sure to call the super implementation of `viewDidUnload`. Next we set all three of our IBOutlet properties to nil.

Sometimes people are confused as to why they need to set their IBOutlet properties to nil in `viewDidUnload`. After all, didn't the view get released by the view controller before this method was even called? Yes, the container view, the one the view controller is connected to via its `view` property in Interface Builder, is indeed released. However, we have three properties pointing to the label, text view, and image view. These properties retain the objects they are connected to, so even though the superview of these items is released, the items themselves won't be released until we also get rid of our relationship to them.

So remember that any IBOutlet properties in the view controller need to be set to nil in the `viewDidUnload` method, and any object that you create in the `viewDidLoad` method also needs to be released. If the view is brought up again later, the connections will be re-made, and `viewDidLoad` will be called again, so we don't lose anything except the risk of being shut down for failing to free up memory.