Your Second iOS App: Storyboards

(Not Recommended)

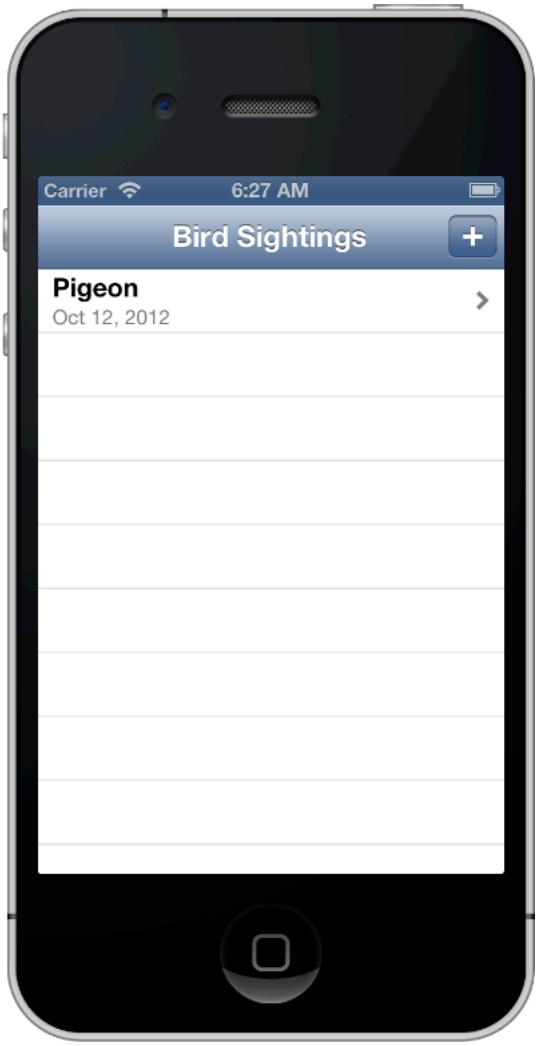
Developer

Contents

About Creating Your Second iOS App

Important This document has not been updated for iOS 7 or later. The content is largely covered in *Start Developing iOS Apps Today (Retired)*.

Your Second iOS App: Storyboards describes how to use storyboards to create a navigation-based app. With the app, users can view details about each item in a master list and add new items to the list. The finished app looks something like this:



After you complete the steps in this tutorial, you'll know how to:

- Design a model layer that represents and manages an app's data
- Create new scenes and segues in a storyboard
- Pass data to and retrieve it from a scene

To get the most from this tutorial, you should already have some familiarity with iOS app programming in general and the Objective-C language in particular. If you've never written an iOS app, read *Start Developing iOS Apps Today (Retired)* (and go through the tutorial *Your First iOS App*) before you begin this tutorial.

At a Glance

Your Second iOS App: Storyboards builds on the knowledge you gained in *Your First iOS App* (which is part of *Start Developing iOS Apps Today (Retired)*): It introduces you to more of the powerful features that storyboards provide and also describes some ways in which you can take advantage of table views in an app.

Designing and Implementing a Model Layer

A well-designed app defines a model layer that includes both the data objects that the app handles and the management of the data objects. Although the model layer in the tutorial app is very simple, designing and implementing it introduces you to concepts that you can apply to more complex apps.

Relevant chapter: "Designing the Model Layer" (page 16)

Designing and Implementing the Master and Detail Scenes

The app in this tutorial is based on the master-detail format, in which users select an item in a master list to see details about the item in a secondary screen. Many iOS apps are based on the master-detail format, including Mail and Settings. This tutorial shows how easy it is to use storyboards to base an app on this format and to send data to the detail screen for display.

Relevant chapters: "Designing the Master Scene" (page 29) and "Displaying Information in the Detail Scene" (page 40)

Creating a New Scene

Most of the Xcode templates place one or more scenes in the storyboard by default, but you can also add new scenes as needed. In this tutorial, you add a scene and embed it within a navigation controller so that you can offer users a way to enter new information to include in the master list.

Relevant chapter: "Enabling the Addition of New Items" (page 54)

Solving Problems and Thinking About Your Next Steps

As you create the app in this tutorial, you might encounter problems that you don't know how to solve. *Your Second iOS App: Storyboards* includes some problem-solving advice, in addition to complete code listings against which you can compare your code.

There are many ways in which you can increase your knowledge of iOS app development. This tutorial suggests several ways to improve the app you create and to learn new skills.

Relevant chapters: "Troubleshooting" (page 85), "Code Listings" (page 90), "Next Steps" (page 87)

See Also

In addition to the documents suggested in "Next Steps" (page 87), there are many other resources that can help you develop great iOS apps:

- To learn about the recommended approaches to designing the user interface and user experience of an iOS app, see *iOS Human Interface Guidelines*.
- For comprehensive guidance on creating a full-featured iOS app, see *iOS App Programming Guide*.
- To learn more about the many features of Xcode, see *Xcode 4 User Guide*.
- To learn about all the tasks you need to perform as you prepare to submit your app to the App Store, see *App Distribution Guide*.

Getting Started

To create the iOS app in this tutorial, you need Xcode 4.5 or later and the iOS SDK for iOS 6.0 or later. When you install Xcode on your Mac, you also get the iOS SDK, which includes the programming interfaces of the iOS platform.

In this chapter, you create a new Xcode project, build and run the default app that the project creates, and learn how the storyboard defines the user interface and some of the app's behavior.

Create a New Project

The app in this tutorial displays a list of bird sightings. When users select a specific sighting, they see some details about it on a new screen. Users can also add to the list by entering details about new sightings.

Transitioning from a master list of items to a more detailed view of one item is part of a common design pattern in iOS apps. Many iOS apps allow users to drill down into a hierarchy of data by navigating through a series of progressively more detailed screens. For example, Mail allows users to drill down from a master list of accounts through progressively more detailed screens until they reveal a specific message.

Xcode provides the Master-Detail template to make it easy to build an app that allows users to navigate through a data hierarchy.

Begin developing the BirdWatching app by creating a new Xcode project for an iOS app that's based on the Master-Detail template.

To create a new project for the BirdWatching app

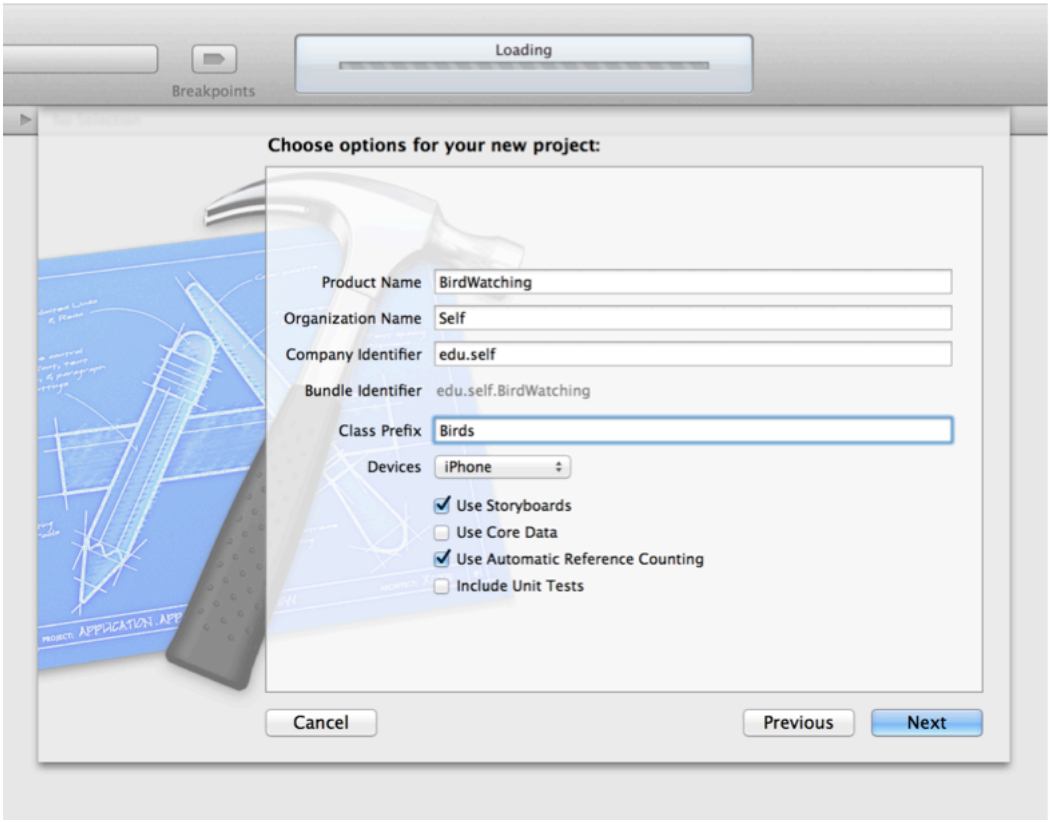
1. Open Xcode and choose File > New > Project.
2. In the iOS section at the left side of the dialog, select Application.
3. In the main area of the dialog, select Master-Detail Application and then click Next.

   A new dialog appears that prompts you to name your app and choose additional options for your project.
4. In the dialog fill in the following fields with these values:
   - **Product Name** `BirdWatching`
   - **Organization Name** Your company's name. If you don't have a company name, you can use `Self`.

- **Company Identifier** The prefix of your company's bundle identifier. If you don't have a company identifier, you can use `edu.self`.
- **Class Prefix** `Birds`

5. In the Devices pop-up menu, make sure that iPhone is chosen.
6. Make sure that the Use Storyboards and Use Automatic Reference Counting options are selected and that the Use Core Data and the Include Unit Tests options are unselected.

   After you finish entering this information, the dialog should look similar to this:



7. Click Next.
8. In the new dialog that appears, enter a location to save your project, make sure the Source Control option is unselected, and click Create.

   Xcode opens your new project in a window, which is called the *workspace window*. By default, the workspace window displays information about the BirdWatching target.

Build and Run the Default Project

As with all template-based projects, you can build and run the new project before you make any changes. It's a good idea to do this because it helps you understand the functionality that the template provides.

To build the default project and run it in iOS Simulator

1. Make sure that the Scheme pop-up menu in the Xcode toolbar displays BirdWatching > iPhone 6.0 Simulator.

   If the menu doesn't display this choice, open it and choose iPhone 6.0 Simulator.

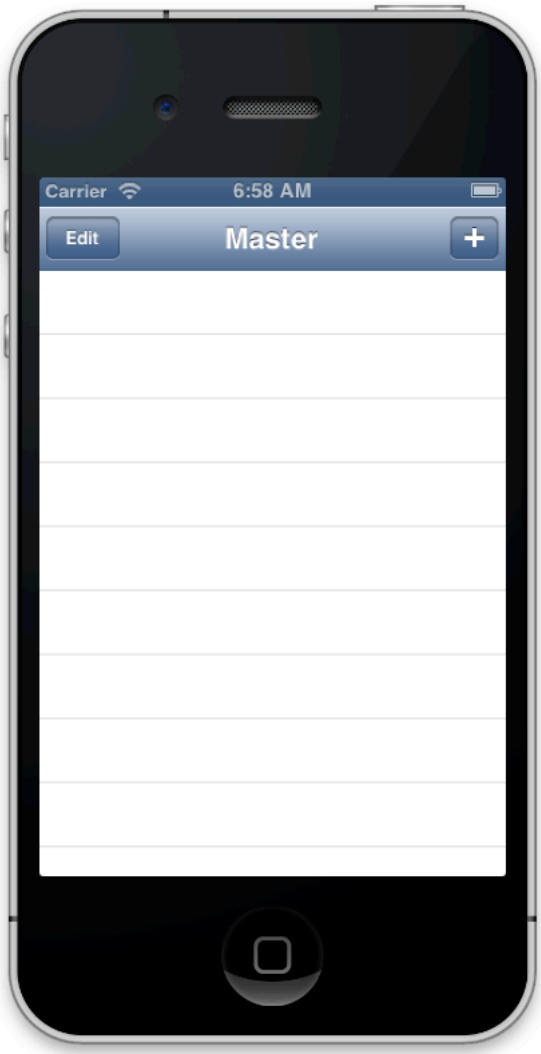
2. Click the Run button in the toolbar (or choose Product > Run).

   If Xcode asks you whether you want to enable developer mode, you can click Enable to let Xcode perform certain debugging tasks without requiring your password every time. For this tutorial, it doesn't matter whether you enable developer mode or not.

   Xcode displays messages about the build process in the activity viewer, which is in the middle of the toolbar.

After Xcode finishes building your project, iOS Simulator should start automatically.

When you created the project, because you specified an iPhone product (as opposed to an iPad product), Simulator displays a window that looks like an iPhone. On the simulated iPhone screen, Simulator opens the default app, which should look like this:



There is already quite a bit of functionality in the default app. For example, the navigation bar includes an Add button (+) and an Edit button that you can use to add and remove items in the master list.

> **Note:** In this tutorial, you won't be using the Edit button, and you'll create the Add button in a way that's different from the way the template creates it. Although you won't use the template-provided code that's associated with these buttons in later steps, leave the buttons in place for now so that you can test the default app.

In Simulator, click the Add button (+) in the right end of the navigation bar to add an item to the list (by default, the item is the current data and time). After you add an item, select it to reveal the detail screen. The default detail screen should look like this:

As the detail screen is revealed, notice that a back button appears in the left end of the navigation bar, titled with the title of the previous screen (in this case, Master). The back button is provided automatically by the navigation controller that manages the master-detail hierarchy. Click the back button to return to the master list screen.

In the next section, you use the storyboard to represent the app's user interface and to implement much of its behavior. For now, quit iOS Simulator.

Examine the Storyboard and Its Scene

A storyboard represents the screens in an app and the transitions between them. The storyboard in the BirdWatching app contains just a few screens, but a more complex app might have multiple storyboards, each of which represents a different subset of its screens.

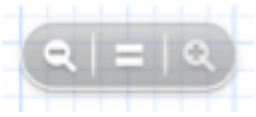
Click `MainStoryboard.storyboard` in the Xcode project navigator to open the BirdWatching app's storyboard on the canvas. The project navigator should already be open in the left side of the Xcode window. If it's not, click the button that looks like this: 

You might have to adjust the zoom level to see all three screens on the canvas without scrolling.

To adjust the canvas zoom level

Do one of the following:

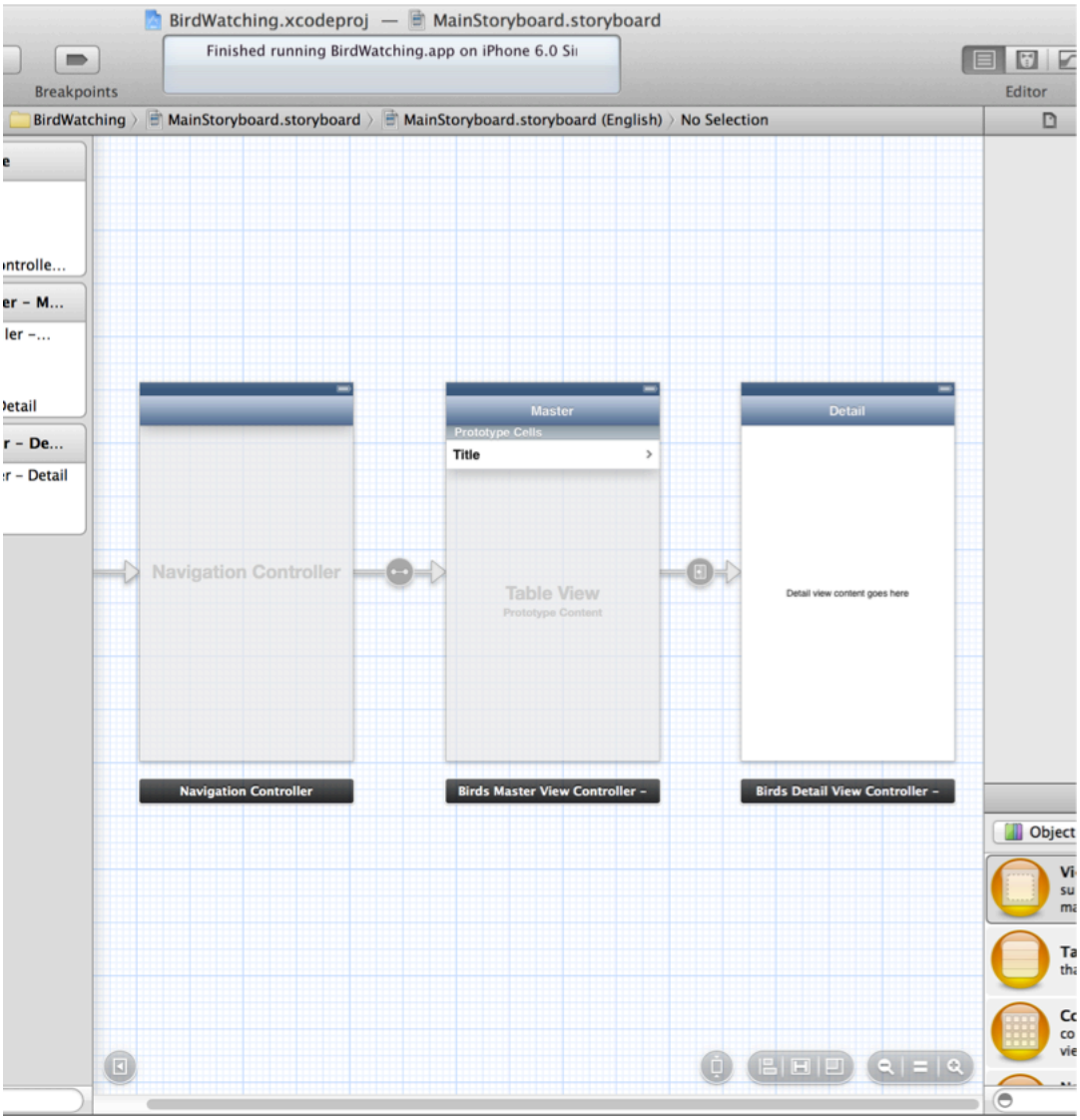
- Click the zoom controls in the lower-right corner of the canvas.

  The zoom controls look like this: 

  The middle zoom control acts as a toggle that switches between the most recent zoom level and 100%.
- Choose Editor > Canvas > Zoom, and then choose the appropriate zoom command.
- Double-click the canvas.

  A double-click on the canvas acts as a toggle that switches between 100% and 50% zoom levels.

Zoom in on the canvas so that you see all three screens in the project. You should see something like this:



The default storyboard in the Master-Detail Application template contains three scenes and two segues.

A **scene** represents an onscreen content area that is managed by a view controller. A **view controller** is an object that manages a set of views that are responsible for drawing content onscreen. (In the context of a storyboard, *scene* and *view controller* are synonymous terms.) The leftmost scene in the default storyboard represents a navigation controller. A navigation controller is called a *container view controller* because, in addition to its views, it also manages a set of other view controllers. For example, the navigation controller in the default BirdWatching app manages the master and detail view controllers, in addition to the navigation bar and the back button that you saw when you ran the app.

A **segue** represents the transition from one scene (called the *source*) to the next scene (called the *destination*). For example, in the default project, the master scene is the source and the detail scene is the destination. When you tested the default app, you triggered a segue from the source to the destination when you selected the Detail item in the master list. In this case, the segue is a **push segue**, which means that the destination scene slides over the source scene from right to left. On the canvas, a push segue looks like this:

When you create a segue between two scenes, you have the opportunity to customize the setup of the destination scene and pass data to it (you do this in "Send Data to the Detail Scene" (page 49)). You learn more about how to send data from a destination scene back to its source in "Get the User's Input" (page 71).

A **relationship** is a type of connection between scenes. On the canvas, a relationship looks similar to a segue, except that its icon looks like this:

In the default storyboard, there is a relationship between the navigation controller and the master scene. In this case, the relationship represents the containment of the master scene by the navigation controller. When the default app runs, the navigation controller automatically loads the master scene and displays the navigation bar at the top of the screen.

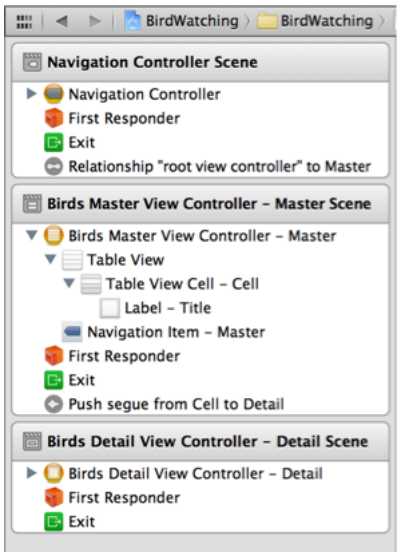
The canvas displays a graphical representation of each scene, its contents, and its connections, but you can also get a hierarchical listing of this information in the document outline. The document outline pane appears between the project navigator and the canvas. If you don't see it, click the Document Outline button in the lower-left corner of the canvas to reveal it. The Document Outline button looks like this:

Tip: Sometimes it's easier to select an element in the document outline than it is to select it on the canvas.

To examine the contents of scenes in the document outline

1. If necessary, resize the document outline pane so that you can read the names of the elements.
2. Click the disclosure triangle next to each element in a view controller section.

   For example, when you reveal the hierarchy of elements in the Birds Master View Controller section, you see something like this:



Recap

In this chapter, you used Xcode to create a new iOS app project based on the Master-Detail Application template. When you ran the default app, you found that it behaved much like other navigation-based iOS apps on iPhone, such as Mail and Contacts.

When you opened the template-provided storyboard on the Xcode canvas, you learned a couple of ways to examine the storyboard and its contents. You also learned about the components of the storyboard and the parts of the app that they represent.

Designing the Model Layer

Every app handles data of some kind. In this tutorial, the BirdWatching app handles a list of bird-sighting events. To follow the Model-View-Controller (MVC) design pattern, you need to create classes that represent and manage this data.

In this chapter, you design two classes. The first represents the primary unit of data that the app handles. The second class creates and manages instances of this data unit. These classes, together with the master list of data, make up the model layer of the BirdWatching app.

> **Note:** The Master-Detail template defines an array containing the placeholder data. You add this data when you click the Add button (+) in the default app. In a later step, you'll replace the template-provided array with the classes that you create next.

Determine the Unit of Data and Create a Data Object Class

To design a data object class, first examine the app's functionality to discover what units of data it needs to handle. For example, you might consider defining a bird class and a bird-sighting class. But to keep this tutorial as simple as possible, you'll define a single data object class—a bird-sighting class—that contains properties that represent a bird name, a sighting location, and a date.

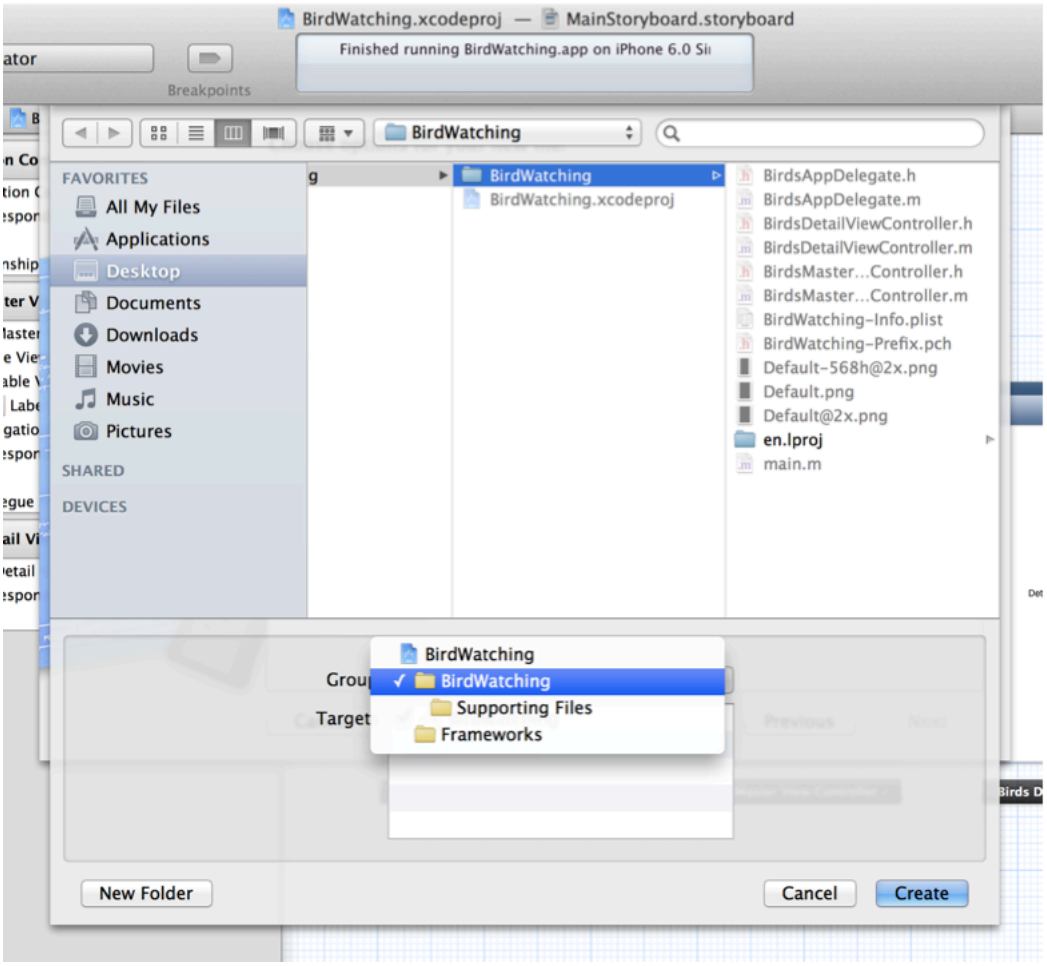
To produce bird-sighting objects that the BirdWatching app can use, you need to add a custom class to your project. Because the bird-sighting object needs very little functionality (for the most part, it simply needs to behave as an Objective-C object), create a new subclass of `NSObject`.

To create the class files for the bird-sighting object

1. In Xcode, choose File > New > File (or press Command-N).
2. In the dialog that appears, select Cocoa Touch in the iOS section at the left side of the dialog.
3. In the main area of the dialog, select Objective-C class and then click Next.
4. In the next pane of the dialog, type the name `BirdSighting` in the Class field, choose `NSObject` in the "Subclass of" pop-up menu, and click Next.

   By convention, the name of a data object class is a noun because it names the thing that the class represents.

5. In the next dialog that appears, choose the BirdWatching folder in the Group pop-up menu:



6. In the dialog, click Create.

   Xcode creates two new files, named `BirdSighting.h` and `BirdSighting.m`, and adds them to the BirdWatching folder in the project. By default, Xcode automatically opens the implementation file (that is, `BirdSighting.m`) in the editor area of your workspace window.

The `BirdSighting` class needs a way to hold the three pieces of information that define a bird sighting. In this tutorial, you use declared properties to represent these items. When you use declared properties, you let the compiler synthesize accessor methods for each one.

The `BirdSighting` class also needs to initialize new instances of itself with information that represents an individual bird sighting. To accomplish this, add a custom initializer method to the class.

First, declare the properties and the custom initialization method in the header file.

To declare properties and a custom initializer method in BirdSighting.h

1. Select `BirdSighting.h` in the project navigator to open it in the editor area.
2. Add the following code between the `@interface` and `@end` statements:

```
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *location;
@property (nonatomic, strong) NSDate *date;
```

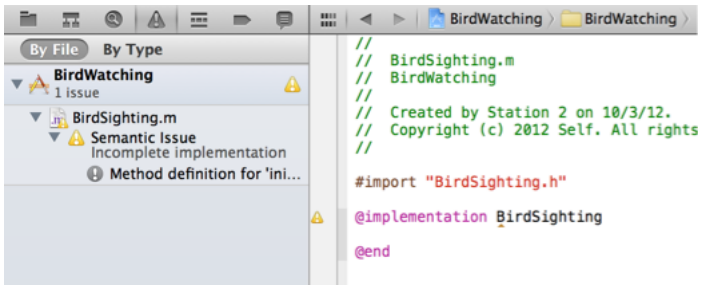
3. Add the following code line after the third property definition:

```
-(id)initWithName:(NSString *)name location:(NSString *)location
date:(NSDate *)date;
```

To implement the custom initializer method

1. In the project navigator, select `BirdSighting.m`.

   When `BirdSighting.m` opens in the editor, Xcode displays a warning icon—in the activity viewer in the middle of the toolbar and in the bar above the editor. Click the warning button in the navigator selector bar to view the warning (the warning button is the middle button in the navigator selector bar). You should see something like this:



   In this case, Xcode is reminding you that the implementation of the `BirdSighting` class is incomplete because you haven't implemented the `initWithName` method. You fix this problem next.

2. In the code editor, between the `@implementation` and `@end` statements in `BirdSighting.m`, begin typing the `initWithName` method and let code completion finish it.

Code completion is especially useful when entering method signatures because it helps you avoid common mistakes, such as mixing the order of parameters. When you start typing `-(id)i` you should see something like this:

When you press Return, Xcode adds the highlighted code-completion suggestion to your file. If the highlighted suggestion is not the one you want, use the Arrow keys to highlight a different one.

A method requires an implementation block—which is not included in the completed method signature—so Xcode displays a couple of problem indicators. You can ignore these indicators because you add the implementation code next.

3. Implement the `initWithName` method by entering the following code after the method signature:

```
{
    self = [super init];
    if (self) {
       _name = name;
       _location = location;
       _date = date;
       return self;
    }
    return nil;
}
```

After you enter all the code in this step, the warning and problem indicators disappear.

Notice that the `initWithName` method uses versions of the property names that begin with an underscore (the `_name`, `_location`, and `_date` symbols refer to the instance variables that these properties represent). By convention, the underscore serves as a reminder that you shouldn't access instance variables directly. In

almost all of your code, you should use the accessor methods—such as `self.name`—to get or set an object's properties. The only two places where you should *not* use accessor methods are in `init` methods—such as `initWithName`—and in `dealloc` methods.

From an academic perspective, avoiding the direct access of instance variables helps preserve encapsulation, but there are also a couple of practical benefits:

- Some Cocoa technologies (notably key-value coding) depend on the use of accessor methods and on the appropriate naming of the accessor methods. If you don't use accessor methods, your app may be less able to take advantage of standard Cocoa features.
- Some property values are created on demand. If you try to use the instance variable directly, you may get `nil` or an uninitialized value. (A view controller's view is a good example of a property value that's created on demand.)

(If you're interested, you can read more about encapsulation in "Storing and Accessing Properties" in *Cocoa Fundamentals Guide*.)

The `BirdSighting` class is one part of the model layer in the BirdWatching app. To preserve the `BirdSighting` class as a pure representation of a model object, you must also design a controller class to instantiate new `BirdSighting` objects and control the collection that contains them. A data controller class allows other objects in the app to access individual `BirdSighting` objects and the master list without needing to know anything about how the data model is implemented. You create the data controller class in the next step.

Create a Data Controller Class

A data controller class typically handles the loading, saving, and accessing of data for an app. Although the BirdWatching app does not access a file to load or store its data, it still needs a data controller to create new `BirdSighting` objects and manage a collection of them. Specifically, the app needs a class that:

- Creates the master collection that holds all `BirdSighting` objects
- Returns the number of `BirdSighting` objects in the collection
- Returns the `BirdSighting` object at a specific location in the collection
- Creates a new `BirdSighting` object using input from the user and adds it to the collection

As you did for the `BirdSighting` class, create the interface and implementation files for a new class that inherits from `NSObject` and add them to the project.

To create the data controller class files

1. Choose File > New > File (or press Command-N).
2. In the dialog that appears, select Cocoa Touch in the iOS section at the left side of the dialog.
3. In the main area of the dialog, select Objective-C class and then click Next.
4. In the next pane of the dialog, enter the class name `BirdSightingDataController`, choose NSObject in the Subclass pop-up menu, and click Next.
5. In the next dialog that appears, choose the BirdWatching folder in the Group pop-up menu and click Create.

In this tutorial, the collection of `BirdSighting` objects is represented by an array. An **array** is a collection object that holds items in an ordered list and includes methods that can access items at specific locations in the list. The BirdWatching app allows users to add new bird sightings to the master list, so you use a mutable array, which is an array that can grow.

The `BirdSightingDataController` class needs a property for the master array that it creates.

To declare the data controller's property

1. Open `BirdSightingDataController.h` in the editor by selecting it in the project navigator.
2. Add the following code line between the `@interface` and `@end` statements:

```
@property (nonatomic, copy) NSMutableArray *masterBirdSightingList;
```

Notice the `copy` attribute in the `masterBirdSightingList` property definition: It specifies that a copy of the object should be used for assignment. Later, in the implementation file, you'll create a custom setter method that ensures that the array copy is also mutable.

Don't close the header file yet because you still need to add the method declarations. As mentioned at the beginning of this section, there are four tasks the data controller needs to perform. Three of these tasks give other objects ways to get information about the list of `BirdSighting` objects or to add a new object to the list. But the "create the master collection" task is a task that only the data controller object needs to know about. Because this method does not need to be exposed to other objects, you do not need to declare it in the header file.

To declare the data controller's three data-access methods

1. Add the following code lines after the property declaration in the `BirdSightingDataController.h` file:

```
- (NSUInteger)countOfList;
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex;
- (void)addBirdSightingWithSighting:(BirdSighting *)sighting;
```

Notice that Xcode displays red problem icons in the gutter to the left of the `objectInListAtIndex` and `addBirdSightingWithSighting` methods. When you click either indicator, Xcode displays the message "Expected a type." This means that Xcode does not recognize `BirdSighting` as a valid return type. To fix this, you need to add a **forward declaration**, which is a statement that tells the compiler to treat a symbol as a class symbol. In effect, it "promises" the compiler that the project contains a definition for this class elsewhere.

2. Add a forward declaration of the `BirdSighting` class.

   Enter the following line between the `#import` and `@interface` statements at the beginning of the header file:

   ```
   @class BirdSighting;
   ```

   After a few seconds, the problem icons disappear.

You're finished with the header file, so now open the `BirdSightingDataController.m` file in the editor and begin implementing the class. Notice that Xcode displays a warning indicator in the gutter next to the `@implementation` statement because you haven't yet implemented the methods you declared in `BirdSightingDataController.h`.

As you determined earlier, the data controller needs to create the master list and populate it with a placeholder item. A good way to do this is to write a method that performs this task, and then call it using the data controller's `init` method.

To implement the list-creation method

1. Import the `BirdSighting` header file so that the data controller methods can refer to objects of this type.

   Add the following code line after the `#import "BirdSightingDataController.h"` statement:

   ```
   #import "BirdSighting.h"
   ```

2. Declare the list-creation method by adding an `@interface` statement before the `@implementation` statement.

The `@interface` statement should look like this:

```
@interface BirdSightingDataController ()
- (void)initializeDefaultDataList;
@end
```

The `@interface BirdSightingDataController ()` code block is called a class extension. A **class extension** allows you to declare a method that is private to the class (to learn more, see "Categories Extend Existing Classes" in *Programming with Objective-C*).

3. Implement the list-creation method by entering the following code lines between the `@implementation` and `@end` statements:

```
- (void)initializeDefaultDataList {
    NSMutableArray *sightingList = [[NSMutableArray alloc] init];
    self.masterBirdSightingList = sightingList;
    BirdSighting *sighting;
    NSDate *today = [NSDate date];
    sighting = [[BirdSighting alloc] initWithName:@"Pigeon"
location:@"Everywhere" date:today];
    [self addBirdSightingWithSighting:sighting];
}
```

The `initializeDefaultDataList` method does the following things: First, it assigns a new mutable array to the `sightingList` variable. Then, it uses some default data to create a new `BirdSighting` object and passes it to the `addBirdSightingWithSighting` method that you declared in "To declare the data controller's three data-access methods" (page 21), which adds the new sighting to the master list.

Although Xcode automatically synthesized accessor methods for the master list property, you need to override its default setter method to make sure that the new array remains mutable. By default, the method signature of a setter method is `set`*PropertyName* (notice that the first letter of the property name becomes capitalized when it's in the setter method name). It's crucial that you use the correct name when you create a custom accessor method; otherwise, the default accessor method is called instead of your custom method.

To implement a custom setter for the master list property

- Add the following code lines within the `@implementation` block of `BirdSightingDataController.m`:


```
- (void)setMasterBirdSightingList:(NSMutableArray *)newList {
    if (_masterBirdSightingList != newList) {
        _masterBirdSightingList = [newList mutableCopy];
    }
}
```

By default, Xcode does not include a stub implementation of the `init` method when you create new Objective-C class files. This is because most objects don't need to do anything other than call `[super init]`. In this tutorial, the data controller class needs to create the master list.

To initialize the data controller object

- Enter the following code lines within the `@implementation` block of `BirdSightingDataController.m`:

```
- (id)init {
    if (self = [super init]) {
        [self initializeDefaultDataList];
        return self;
    }
   return nil;
}
```

This method assigns to `self` the value returned from the super class's initializer. If `[super init]` is successful, the method then calls the `initializeDefaultDataList` method that you wrote earlier and returns the newly initialized instance of itself.

You've given the data controller class the ability to create the master list, populate it with a placeholder item, and initialize a new instance of itself. Now give other objects the ability to interact with the master list by implementing the three data-access methods you declared in the header file:

- `countOfList`
- `objectInListAtIndex:`
- `addBirdSightingWithSighting:sighting:`

To implement the data controller's data-access methods

1. Implement the `countOfList` method by entering the following code lines in the `@implementation` block of `BirdSightingDataController.m`:

```
- (NSUInteger)countOfList {
    return [self.masterBirdSightingList count];
}
```

   The `count` method is an `NSArray` method that returns the total number of items in an array. Because `masterBirdSightingList` is of type `NSMutableArray`, which inherits from `NSArray`, the property can respond to the `count` message.

2. Implement the `objectInListAtIndex:` method by entering the following code lines:

```
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex {
    return [self.masterBirdSightingList objectAtIndex:theIndex];
}
```

3. Implement the `addBirdSightingWithSighting:sighting:` method by entering the following code lines:

```
- (void)addBirdSightingWithSighting:(BirdSighting *)sighting {
    [self.masterBirdSightingList addObject:sighting];
}
```

   This method creates and initializes a new `BirdSighting` object by sending to the `initWithName:location:date:` method the name and location the user entered, along with today's date. Then, the method adds the new `BirdSighting` object to the array.

When you finish entering the code in this step, you complete the implementation of `BirdSightingDataController` and Xcode removes the warning indicator.

You can build and run the app at this point, but nothing has changed from the first time you ran it because the view controllers don't know anything about the data model you implemented. In the next chapter, you edit the master view controller and app delegate files so that the app can display the placeholder data.

Recap

In this chapter, you designed and implemented the data layer for the BirdWatching app. Following the MVC design pattern, you created classes that contain and manage the data that the app works with.

At this point in the tutorial, the project should contain interface and implementation files for both the `BirdSighting` and the `BirdSightingDataController` classes. The code for all four files is listed below so that you can check the code in your project for accuracy.

The code in `BirdSighting.h` should look like this:

```
#import <Foundation/Foundation.h>


@interface BirdSighting : NSObject

@property (nonatomic, copy) NSString *name;

@property (nonatomic, copy) NSString *location;

@property (nonatomic, strong) NSDate *date;

-(id)initWithName:(NSString *)name location:(NSString *)location date:(NSDate
*)date;

@end
```

The code in `BirdSighting.m` should look like this:

```
#import "BirdSighting.h"


@implementation BirdSighting

-(id)initWithName:(NSString *)name location:(NSString *)location date:(NSDate
*)date

{

    self = [super init];

    if (self) {

        _name = name;

        _location = location;

        _date = date;

        return self;

    }

    return nil;

}
```



```
@end
```

The code in `BirdSightingDataController.h` should look like this:

```
#import <Foundation/Foundation.h>
@class BirdSighting;
@interface BirdSightingDataController : NSObject
@property (nonatomic, copy) NSMutableArray *masterBirdSightingList;
- (NSUInteger)countOfList;
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex;
- (void)addBirdSightingWithSighting:(BirdSighting *)sighting;
@end
```

The code in `BirdSightingDataController.m` should look like this:

```
#import "BirdSightingDataController.h"
#import "BirdSighting.h"
@interface BirdSightingDataController ()
- (void)initializeDefaultDataList;
@end
@implementation BirdSightingDataController
- (void)initializeDefaultDataList {
    NSMutableArray *sightingList = [[NSMutableArray alloc] init];
    self.masterBirdSightingList = sightingList;
    BirdSighting *sighting;
    NSDate *today = [NSDate date];
   sighting = [[BirdSighting alloc] initWithName:@"Pigeon" location:@"Everywhere"
 date:today];
    [self addBirdSightingWithSighting:sighting];
}
- (void)setMasterBirdSightingList:(NSMutableArray *)newList {
    if (_masterBirdSightingList != newList) {
        _masterBirdSightingList = [newList mutableCopy];
    }
}
- (id)init {
```



```
    if (self = [super init]) {
        [self initializeDefaultDataList];
        return self;
    }
    return nil;
}
- (NSUInteger)countOfList {
    return [self.masterBirdSightingList count];
}
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex {
    return [self.masterBirdSightingList objectAtIndex:theIndex];
}
- (void)addBirdSightingWithSighting:(BirdSighting *)sighting {
    [self.masterBirdSightingList addObject:sighting];
}
@end
```


Designing the Master Scene

When users start the BirdWatching app, the first screen they see displays the master list of bird sightings. In this chapter, you design the appearance of the master list and give the master view controller access to the data in the model layer.

Design the Master View Controller Scene

You've been writing a lot of code so far in this tutorial; now you'll do some design work on the canvas. Open the storyboard on the canvas by selecting `MainStoryboard.storyboard` in the project navigator.

iOS apps often use table views to display lists of items, because tables provide efficient and customizable ways to display both large and small amounts of data. For example, Mail, Settings, Contacts, and Music use various types of tables to display information. In this section, you specify the overall appearance of the master scene and design the layout of its table rows.

To specify the appearance of the master scene

1. On the canvas, adjust the zoom level (if necessary) to focus on the master scene.
2. Double-click the title in the navigation bar (that is, Master) and type `Bird Sightings`.
3. Click the center of the scene to select the table view.

   If you prefer, you can select the table view by selecting Table View in the Birds Master View Controller section in the document outline pane.
4. If necessary, click the Utilities View button to open the utilities area.

   The Utilities View button looks like this: 
5. In the utility area, click the Attributes button to open the Attributes inspector.

   The Attributes button looks like this: 
6. In the Table View section of the Attributes inspector, ensure that the Style pop-up menu displays Plain.

As you can see in the master scene, a row in a plain-style table extends the full width of the table view. The rows in a grouped-style table are inset from the edges of the table view, leaving a margin of the background appearance visible on all sides. (To see an example of a grouped-style table, open the Settings app on an iOS-based device.)

When you use storyboards, you have two convenient ways to design the appearance of a table's rows (which are called *cells*):

- Dynamic prototypes allow you to design one cell and then use it as the template for other cells in the table. Use a dynamic prototype when you want every cell to use the same layout and you can't predict how many cells the table might display.
- Static cells allow you to design the overall layout of the table, including the total number of cells. Use static cells when a table does not change its appearance, regardless of the specific information it displays.

In the master scene, you want each bird-sighting item to be displayed using the same layout, regardless of how many items the user adds, so you design a prototype cell for the master scene's table. As the app runs, copies of this prototype cell are created as needed to display bird-sighting items.

For this step, the canvas should still be open and the Attributes inspector should still be focused on the master scene's table view.

To design a prototype cell for the master bird-sighting list

1. In the Table View section of the Attributes inspector, make sure that Dynamic Prototypes is chosen in the Content pop-up menu.
2. On the canvas, select the table view cell to display table-cell information in the Attributes inspector.
3. In the Table View Cell section of the Attributes inspector, choose Subtitle in the Style pop-up menu.

   The built-in subtitle style causes the cell to display two left-aligned labels: Title in a large bold font and Subtitle in a smaller gray font. When you use a built-in cell style, the connections between these labels and the cell's properties are made automatically. In a cell that uses the subtitle style, the `textLabel` property refers to the title and the `detailTextLabel` property refers to the subtitle.

4. Change the default value in the Identifier text field of the Table View Cell Attributes inspector.

   The value in the Identifier field is called a *reuse identifier*, and it gives the compiler a way to identify the appropriate prototype cell when it's time to create new cells for the table. By convention, a cell's reuse identifier should describe what the cell contains, so in this tutorial replace the default value (that is, `Cell`) with `BirdSightingCell`. Note that you'll need to use this reuse ID elsewhere in your code, so you might want to copy it.

5. Still in the Attributes inspector, choose Disclosure Indicator in the Accessory pop-up menu, if necessary.

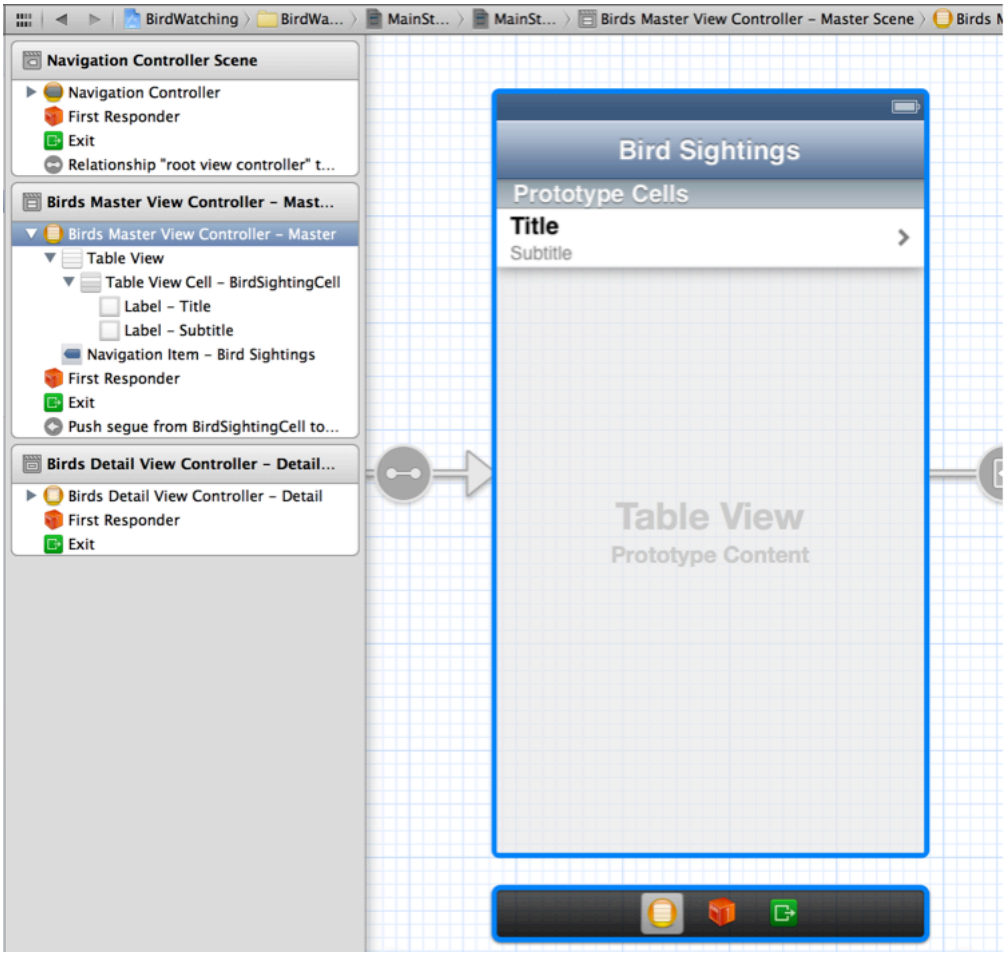
An accessory is a standard user interface (UI) element that can appear in a table cell. The disclosure indicator accessory (which looks similar to ">") tells users that tapping an item reveals related information in a new screen.

Before you edit the master scene's code files, make sure that the scene on the canvas represents the master view controller class in your code. If you forget this step, none of your customization of the code and of the canvas will be visible when you run the app.

To specify the identity of the master scene

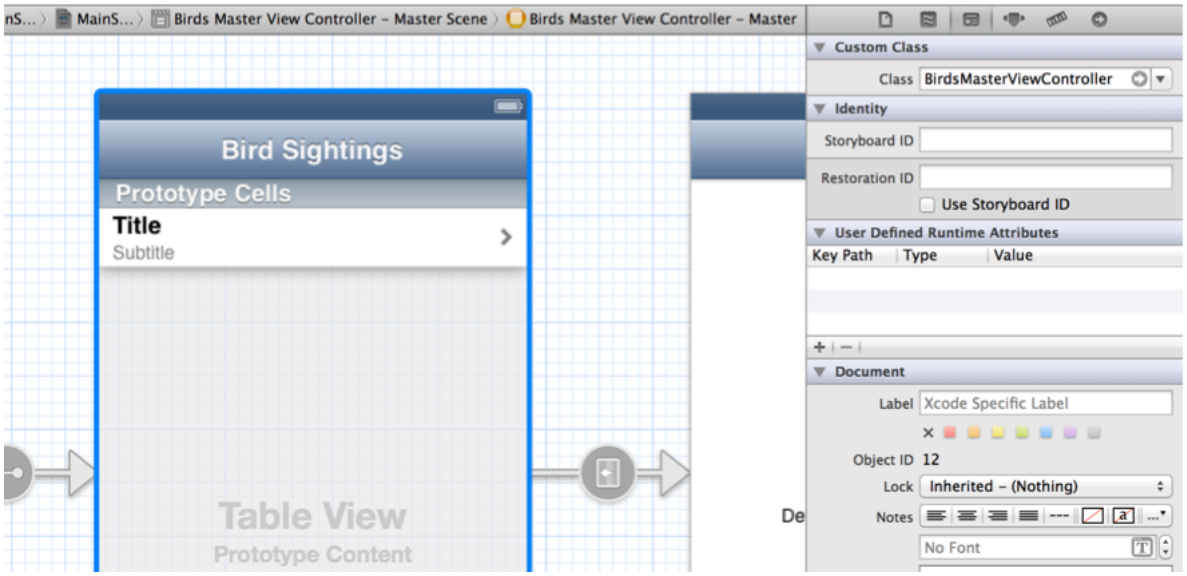
1. In the document outline, select Birds Master View Controller - Master.

   On the canvas, the selected master scene is outlined in blue, as shown here:



2. Click the Identity button at the top of the utility area to open the Identity inspector.
3. In the Identity inspector, make sure that the Class field contains `BirdsMasterViewController`.

You should see something like this:



Clean Up the Master View Controller Implementation File

Before you write the code that allows the master view controller to display the master list of bird sightings, you need to bypass some of the code that the template provides. For example, you don't need the template-provided Edit button (because the master list in the BirdWatching app doesn't support editing) or the default `_objects` array (because you'll use the data model classes you created earlier).

Although you can delete the unnecessary code, it's often better to use the multiline comment symbols to make the code invisible to the compiler (the multiline comment symbols are `/*` and `*/`). Commenting out code that you don't need makes it easier to change the implementation of your app in the future. For example, the Master-Detail template comments out the `moveRowAtIndexPath` and `canMoveRowAtIndexPath` table view methods in the master view controller implementation file because not all apps need tables that can be rearranged.

To bypass unnecessary code in the master view controller implementation file

1. In the project navigator, select `BirdsMasterViewController.m`.
2. Comment out the declaration of the private `_objects` array.

   Because you designed your own data model classes, you don't need the template-provided `_objects` array. After you add the comment symbols, the beginning of `BirdsMasterViewController.m` should look like this:


```
#import "BirdsMasterViewController.h"
/*
@interface BirdsMasterViewController () {
   NSMutableArray *_objects;
}
@end
*/
@implementation BirdsMasterViewController
```

After you add the comment symbols, Xcode displays several problem indicators because the template-provided master view controller code refers to `_objects` in several places. You'll fix some of these problems when you comment out more code. You'll fix the remaining problems when you write your own method implementations.

3. Comment out most of the contents of the `viewDidLoad` method.

   For now, the `viewDidLoad` method should only contain the call to `[super viewDidLoad]`. After you comment out the rest of the template-provided code, your `viewDidLoad` method should look like this:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a
 nib.
    /*
    self.navigationItem.leftBarButtonItem = self.editButtonItem;


    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(insertNewObject:)];
    self.navigationItem.rightBarButtonItem = addButton;
     */
}
```

4. Comment out the `insertNewObject` method.

The `insertNewObject` method adds new objects to the `_objects` array and updates the table. You don't need this method, because you'll be updating your custom data model objects and the master list in other ways.

Xcode removes a few problem indicators when you comment out the `insertNewObject` method.

5. Change the return value of the `canEditRowAtIndexPath` table view method.

   By default, the Master-Detail template enables table view editing in the master view controller's table view. Because you won't be giving users an Edit button in the master scene, change the default return value to `NO`.

   After you make this change, the `canEditRowAtIndexPath` method should look like this:

```
- (BOOL)tableView:(UITableView *)tableView
canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Return NO if you do not want the specified item to be editable.
    return NO;
}
```

6. Comment out the `tableView:commitEditingStyle:forRowAtIndexPath:` method.

   You won't be using the template-provided Edit button in this tutorial, so you don't need to implement the `commitEditingStyle` method's support for removing and adding table rows.

   Commenting out the `commitEditingStyle` method removes one of the problem indicators.

After you finish commenting out the code you don't need (and updating the `canEditRowAtIndexPath` method), you still see a few problem indicators. You fix two of these problems in the next step.

Implement the Master View Controller

Typically, an app displays data in a table by implementing the table view data source methods in a table view controller class. At a minimum, the table view controller class implements the `numberOfRowsInSection` and `cellForRowAtIndexPath` methods.

To display the master list of bird sightings, the master view controller implements these two data source methods. But first, the master view controller must have access to the data that is managed by the model layer objects.

To give the master view controller access to the data in the model layer

1. In the project navigator, select `BirdsMasterViewController.h`.
2. Add a forward declaration of the `BirdSightingDataController` class.

   Between the `#import` and `@interface` statements, add the following code line:

   ```
   @class BirdSightingDataController;
   ```

3. Declare a data controller property.

   Edit the `BirdsMasterViewController` header file so that it looks like this:

   ```
   @interface BirdsMasterViewController : UITableViewController
   @property (strong, nonatomic) BirdSightingDataController *dataController;
   @end
   ```

4. In the project navigator, select `BirdsMasterViewController.m`.
5. In `BirdsMasterViewController.m`, add the following code lines after `#import "BirdsDetailViewController.h"` to import the header files of the model layer classes:

   ```
   #import "BirdSightingDataController.h"
   #import "BirdSighting.h"
   ```

6. Implement the `awakeFromNib` method to associate a new data controller object with the `dataController` property you declared.

   To the `awakeFromNib` method, add the following code line after `[super awakeFromNib];`:

   ```
   self.dataController = [[BirdSightingDataController alloc] init];
   ```

Now that the master view controller has access to the data from the model layer, it can pass that data to the table view data source methods. Make sure that `BirdsMasterViewController.m` is still open for this step.

To implement the table view data source methods in the master view controller

1. Implement the `numberOfRowsInSection` method to return the number of `BirdSighting` objects in the array.

   Replace the default implementation of `numberOfRowsInSection` with the following code:


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

Providing this implementation of the `numberOfRowsInSection` method removes one of the problem indicators, because the method no longer refers to the `_objects` array.

2. Implement the `cellForRowAtIndexPath` method to create a new cell from the prototype and populate it with data from the appropriate `BirdSighting` object.

   Replace the default implementation of `cellForRowAtIndexPath` with the following code:

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


    static NSString *CellIdentifier = @"BirdSightingCell";


    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];


    BirdSighting *sightingAtIndex = [self.dataController
objectInListAtIndex:indexPath.row];
    [[cell textLabel] setText:sightingAtIndex.name];
    [[cell detailTextLabel] setText:[formatter stringFromDate:(NSDate
*)sightingAtIndex.date]];
    return cell;
}
```

   Because the new method implementation does not refer to the `_objects` array, Xcode removes another problem indicator.

A table view object invokes the `cellForRowAtIndexPath` method every time it needs to display a table row. In this implementation of the `cellForRowAtIndexPath` method, you identify the prototype cell that should be used to create new table cells and you specify the format to use for displaying the date. Then, if there are no reusable table cells available, you create a new cell based on the prototype cell you designed on the canvas. Finally, you retrieve the `BirdSighting` object that's associated with the row that the user tapped in the master list and update the new cell's labels with the bird sighting information.

After you complete these steps, the remaining problems are in the `prepareForSegue` method. You'll edit this method later.

Recap

In this chapter, you took advantage of dynamic prototype cells to design the layout of the rows in the master scene's list. Then you implemented the master scene view controller by giving it access to the data in the model layer and using that information in the implementation of the table view data source methods.

At this point in the tutorial, the code in the `BirdsMasterViewController.h` file should look similar to this:

```
#import <UIKit/UIKit.h>
@class BirdSightingDataController;
@interface BirdsMasterViewController : UITableViewController
@property (strong, nonatomic) BirdSightingDataController *dataController;
@end
```

The code in the `BirdsMasterViewController.m` file should look similar to this (methods that you commented out or that you don't edit in this tutorial are not shown):

```
#import "BirdsMasterViewController.h"

#import "BirdsDetailViewController.h"
#import "BirdSightingDataController.h"
#import "BirdSighting.h"

@implementation BirdsMasterViewController

- (void)awakeFromNib
{
```



```
    [super awakeFromNib];
    self.dataController = [[BirdSightingDataController alloc] init];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

#pragma mark - Table View

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

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [self.dataController countOfList];
}

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

    static NSString *CellIdentifier = @"BirdSightingCell";

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
```



```
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];

    BirdSighting *sightingAtIndex = [self.dataController
objectInListAtIndex:indexPath.row];
    [[cell textLabel] setText:sightingAtIndex.name];
    [[cell detailTextLabel] setText:[formatter stringFromDate:(NSDate
*)sightingAtIndex.date]];
    return cell;
}

- (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath
*)indexPath
{
    // Return NO if you do not want the specified item to be editable.
    return NO;
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"showDetail"]) {
        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
        NSDate *object = _objects[indexPath.row];
        [[segue destinationViewController] setDetailItem:object];
    }
}

@end
```

Displaying Information in the Detail Scene

When iOS users tap a table row that includes a disclosure indicator, they expect to see a new screen that displays information related to the row item, typically in another table. In the BirdWatching app, each cell in the master scene's table displays a bird-sighting item and a disclosure indicator. When users tap an item, a detail screen appears that displays the bird's name and location and date of the bird sighting.

In this chapter, you create the detail scene that displays the information about the selected bird sighting.

Edit the Detail View Controller Code

The detail scene should display the information from the `BirdSighting` object that's associated with the selected table cell. Unlike the work you did on the master scene, you'll write the detail view controller code first and design the scene on the canvas in a later step.

To customize the detail view controller header file

1. Select `BirdsDetailViewController.h` in the project navigator.
2. Between the `#import` and `@interface` statements, add a forward declaration of the `BirdSighting` class:

```
@class BirdSighting;
```

3. Change the parent class of the default detail view controller to `UITableViewController`.

   After you do this, the edited `@interface` statement should look like this:

```
@interface BirdsDetailViewController : UITableViewController
```

   Note that changing the default detail scene from a generic view controller to a table view controller is a design decision; that is, this change is not required for the app to run correctly. The main reason for this design decision is that using a table to display the bird sighting details provides a user experience that is consistent with the master scene. A secondary reason is that it gives you the opportunity to learn how to use static cells to design a table.

4. Declare properties that refer to the `BirdSighting` object and its properties.

   You don't need the default `detailItem` and `detailDescriptionLabel` property declarations provided by the template. Replace these with the following property declarations:

```
@property (strong, nonatomic) BirdSighting *sighting;
@property (weak, nonatomic) IBOutlet UILabel *birdNameLabel;
@property (weak, nonatomic) IBOutlet UILabel *locationLabel;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;
```

   Xcode displays additional problem indicators after you replace the default properties because the code still references the old properties.

In the detail scene implementation file, you first need to give the detail scene access to objects of type `BirdSighting`. Then, you need to implement two of the default methods.

To give the detail scene access to BirdSighting objects

1. Select `BirdsDetailViewController.m` in the project navigator.

   Xcode displays red problem indicators next to each usage of the `detailItem` and `detailDescriptionLabel` symbols because you removed the property declarations for them.

2. At the beginning of the file, import the `BirdSighting` header file by adding the following code line:

```
#import "BirdSighting.h"
```

By default, the `BirdsDetailViewController` implementation file includes stub implementations of the `setDetailItem` and `configureView` methods. Notice that the `setDetailItem` method is a custom setter method for the `detailItem` property that was provided by the template (and that you removed in "To customize the detail view controller header file" (page 40)). The reason the template uses a custom setter method—and not the default setter that Xcode can synthesize—is so that `setDetailItem` can call `configureView`.

In this tutorial, the detail item is a `BirdSighting` object, so the `sighting` property takes the place of the `detailItem` property. Because these two properties play similar roles, you can use the structure of the `setDetailItem` method to help you create a custom setter for the `sighting` property.

To create a custom setter method for the sighting property

- In `BirdsDetailViewController.m`, replace the `setDetailItem` method with the following code:


```
- (void)setSighting:(BirdSighting *) newSighting
{
    if (_sighting != newSighting) {
        _sighting = newSighting;

        // Update the view.
        [self configureView];
    }
}
```

The `configureView` method (which is called in both the new `setSighting` method and the default `viewDidLoad` method) updates the UI of the detail scene with specific information. Edit this method so that it updates the detail scene with data from the selected `BirdSighting` object.

To implement the configureView method

- In `BirdsDetailViewController.m`, replace the default `configureView` method with the following code:

```
- (void)configureView
{
    // Update the user interface for the detail item.
   BirdSighting *theSighting = self.sighting;

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
    if (theSighting) {
        self.birdNameLabel.text = theSighting.name;
        self.locationLabel.text = theSighting.location;
        self.dateLabel.text = [formatter stringFromDate:(NSDate
*)theSighting.date];
    }
```



```
}
```

In the next section, you'll lay out the detail scene on the canvas.

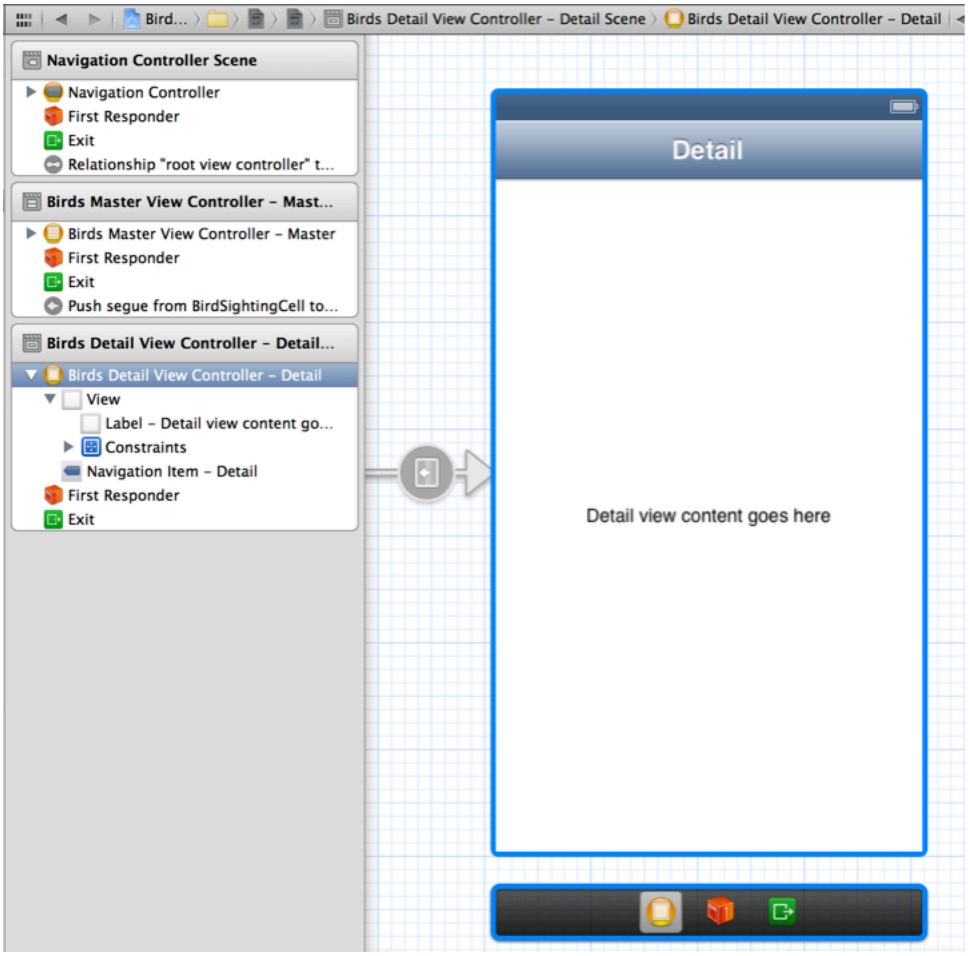
Design the Detail Scene

In the previous section, you changed the parent class of the template-provided detail view controller from `UIViewController` to `UITableViewController`. In this section, you replace the default detail scene on the canvas with a new table view controller from the object library.

To replace the default UIViewController scene with a UITableViewController scene

1. Select `MainStoryboard.storyboard` in the project navigator to open it on the canvas.
2. Select the detail scene and press Delete.

To make sure that you delete the scene itself—and not just an element in the scene—you need to make sure that the entire scene is selected on the canvas. An easy way to do this is to select the scene's view controller in the document outline. For example, when you select Birds Detail View Controller - Detail in the document outline, you see something like this:



3. Drag a table view controller from the object library to the canvas.
4. With the new scene still selected on the canvas, click the Identity button in the utility area to open the Identity inspector.

   If necessary, select Table View Controller in the document outline to ensure that the scene is selected on the canvas.
5. In the Custom Class section of the Identity inspector, choose `BirdsDetailViewController` in the Class pop-up menu.

When you delete a scene from the canvas, all segues to that scene also disappear. You need to reestablish the segue from the master scene to the detail scene so that the master scene can transition to the detail scene when the user selects an item. Make sure that you can see both the master scene and the new detail scene on the canvas.

To create a segue from the master scene to the detail scene

1. Control-drag from the table cell in the master scene to the detail scene.

   In the Selection Segue area of the translucent panel that appears, select Push. A selection segue occurs when the user selects the table row; a push segue causes the new scene to slide over the previous scene from the right edge of the screen. (The Accessory Action area lists segues that can be triggered when the user interacts with a table accessory, such as a detail disclosure button.)

   Notice that Xcode automatically displays a navigation bar in the detail scene. This is because Xcode knows that the source scene—in this case, the master Bird Sightings scene—is part of a navigation controller hierarchy, and so it simulates the appearance of the bar in the detail scene to make it easier for you to design the layout.

2. On the canvas, select the push segue you just created.
3. In the Attributes inspector, enter a custom ID in the Identifier field.

   By convention, it's best to use an identifier that describes what the segue does. In this tutorial, use `ShowSightingDetails` because this segue reveals sighting details in the detail scene.

   If a scene can transition to different destination scenes, it's important to give each segue a unique identifier so that you can differentiate them in code. In this tutorial, the master scene can transition to the detail scene and to the add scene (which you'll create in "Enabling the Addition of New Items" (page 54)), so you need to have a way to distinguish between these two segues.

Although the details about a bird sighting vary, the app should always display them in the same format—specifically: name, date, and location. Because you never want the detail table to display information in a different layout, you can use static table cells to design this layout on the canvas.

By default, a new table view controller scene contains a table that uses prototype-based cells. Before you design the detail table layout, change its content type.

To change the detail table-view content type

1. On the canvas, select the table view in the detail scene.
2. In the Attributes inspector, choose Static Cells in the Content pop-up menu.

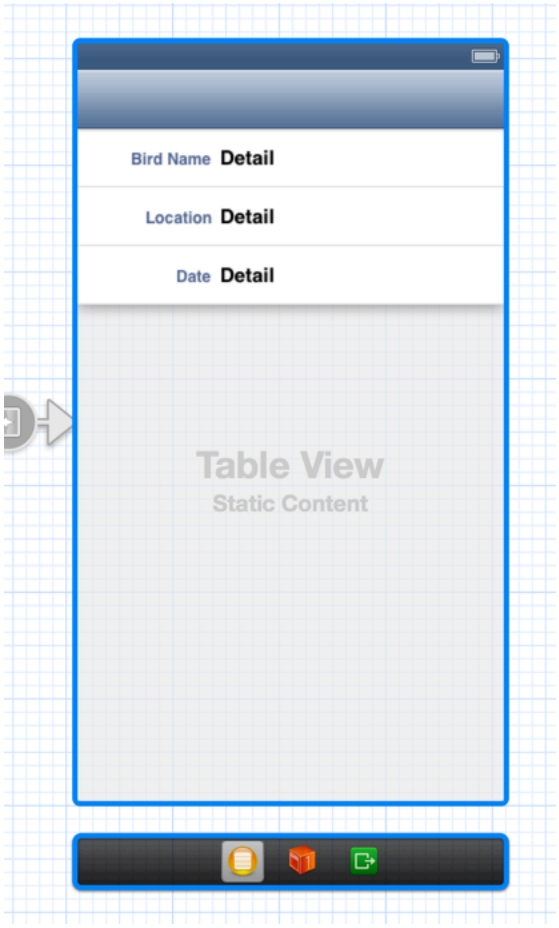
When you change the content type of a table view from dynamic prototypes to static cells, the resulting table automatically contains three cells. By coincidence, three happens to be the number of table rows that you want to display in the app. You could design each cell separately, but because each cell uses the same layout, it's more convenient to design one of them and copy it to create the others.

To design one static table cell and copy it

1. Remove two of the cells by selecting them and pressing Delete.
2. Select the remaining cell (if necessary) and choose `Left Detail` in the Style pop-up menu of the Table View Cell Attributes inspector.
3. In the document outline, select Table View Section.
4. In the Table View Section area of the Attributes inspector, use the Rows stepper to increase the number of rows to 3.
5. In each cell, double-click the title label and enter the appropriate description.

   In the top cell, enter `Bird Name`; in the middle cell, enter `Location`; and in the bottom cell, enter `Date`.

After you finish laying out the cells in the table, the detail scene should look similar to this:



When the app runs, you want the right-hand label in each cell to display one of the pieces of information in the selected `BirdSighting` object. Earlier, you gave the detail view controller three properties, each of which accesses one property of a `BirdSighting` object. In this step, connect each label in the detail scene to the appropriate property in the detail view controller.

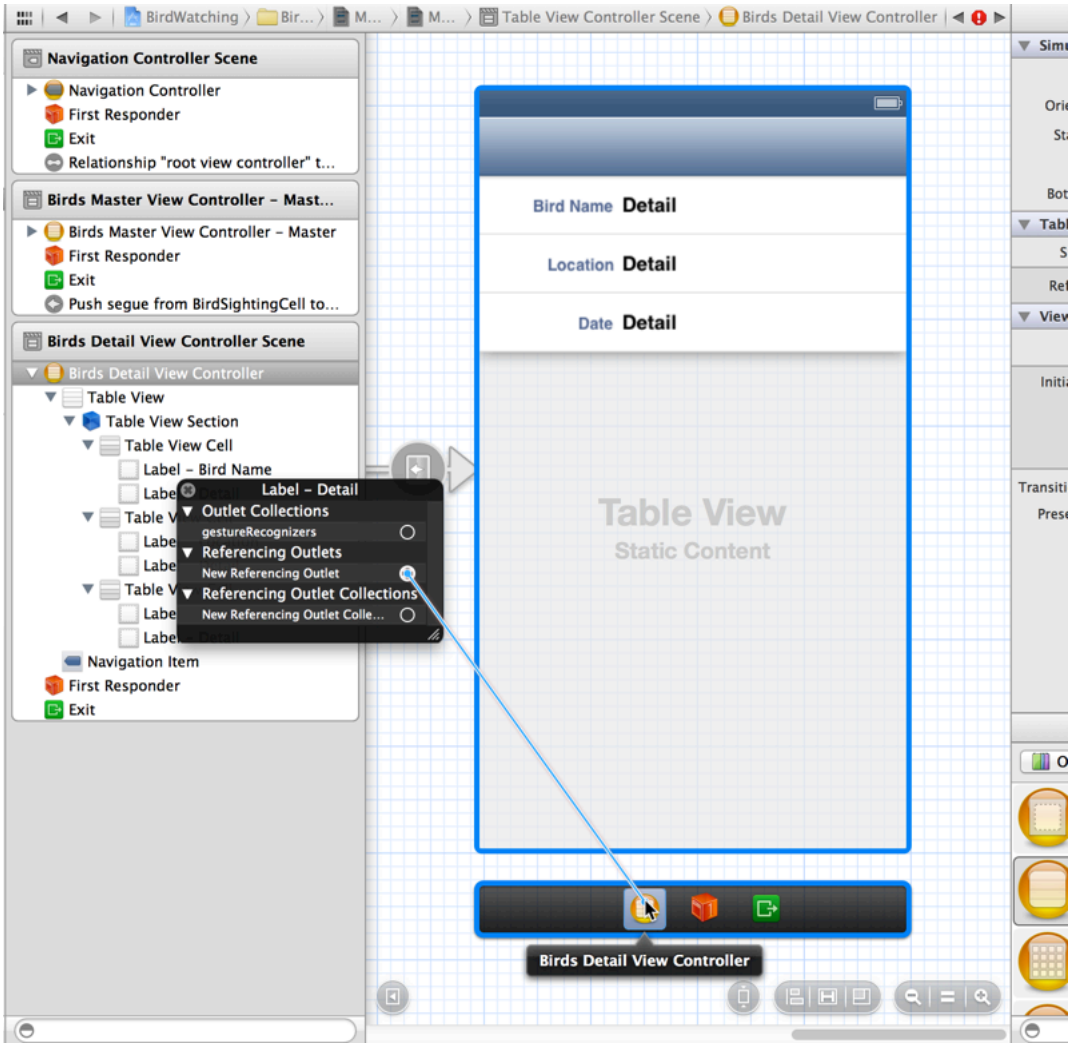
To connect the detail text labels to the detail view controller's properties

1. In the Table View section of the document outline, locate the table view cell that contains the Bird Name label (it should be the first one).
2. Control-click the Label - Detail object listed below Label - Bird Name.
3. In the translucent panel that appears, Control-drag from the empty circle in the New Referencing Outlet item to the `BirdsDetailViewController` object in the scene dock on the canvas.

   The scene dock is the bar below the scene. When the scene or any element within it is selected, the scene dock typically displays three proxy objects: An orange cube that represents the first responder, a yellow sphere that represents the scene's view controller object, and a green square that represents

the destination of an unwind segue. An **unwind segue** is a segue you use to return to an existing scene without instantiating a new view controller object. (Later, you'll be using an unwind segue to add to the master list information about a new bird sighting.) At other times, the scene dock displays the scene's name.

As you Control-drag from the New Referencing Outlet item in the translucent panel, you see something like this:



4. In the Connections panel that appears when you release the Control-drag, choose `birdNameLabel`.
5. Perform steps 1, 2, and 3 with the detail label in the Location cell. This time, choose `locationLabel` in the Connections panel.
6. Perform steps 1, 2, and 3 with the detail label in the Date cell. This time, choose `dateLabel` in the Connections panel.

All the elements of the detail scene's UI seem to be connected with the code, but the detail scene still doesn't have access to the `BirdSighting` object that represents the item the user selected in the master list. You fix this in the next step.

Send Data to the Detail Scene

Storyboards make it easy to pass data from one scene to another using the `prepareForSegue` method. This method is called when the first scene (the *source*) is about to transition to the next scene (the *destination*). The source view controller can implement `prepareForSegue` to perform setup tasks, such as passing to the destination view controller the data it should display in its views.

> **Note:** The default implementation of `prepareForSegue` that's provided by the template uses the default `setDetailItem` method in the detail view controller. Because you replaced `setDetailItem` with `setSighting` (in "To create a custom setter method for the sighting property" (page 41)), you might see a problem indicator telling you that `setDetailItem` has no implementation. You'll fix this problem when you implement the `prepareForSegue` method in the next step.

In your implementation of the `prepareForSegue` method, you need the ID that you gave to the segue between the master scene and the detail scene. In this tutorial, the ID is already part of the code listing for the method; when you write an app from scratch, you need to copy the ID from the segue Attributes inspector.

To send setup information to the detail scene

1. Select `BirdsMasterViewController.m` in the project navigator to open the file in the editor.
2. Make sure the detail view's header file is imported.

   Near the top of the file, you should see the following code line:

   ```
   #import "BirdsDetailViewController.h"
   ```

3. In the `@implementation` block, replace the default `prepareForSegue` method with the following code:

   ```
   - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
       if ([[segue identifier] isEqualToString:@"ShowSightingDetails"]) {
           BirdsDetailViewController *detailViewController = [segue
   destinationViewController];
   ```



```
        detailViewController.sighting = [self.dataController
objectInListAtIndex:[self.tableView indexPathForSelectedRow].row];
    }
}
```

After a few seconds, Xcode removes all the remaining problem indicators.

Build and run the project. In Simulator, you should see the placeholder data that you created in the `BirdSightingDataController` class displayed in the master scene. Notice that the master scene no longer displays the Edit button and Add button (+) because you removed the code that created these buttons from `BirdsMasterViewController.m`.

In the master scene, select the placeholder item. The master scene transitions to the detail scene, which displays information about the item in the table configuration that you designed. Click the back button that appears in the upper-left corner of the screen to return to the master list.

In the next chapter, you'll design a new scene in which user can enter information about a new bird sighting and add it to the master list. For now, quit iOS Simulator.

Recap

In this chapter, you customized the template-provided detail view controller so that it displays the details about the item the user selects in the master list. You edited the detail view controller code to change the parent class to `UITableViewController`, added properties that refer to bird-sighting details, and implemented methods that update the UI.

On the canvas, you replaced the template-provided detail scene with a table view controller. When you did this, you learned that you had to re-create the segue from the master scene to the detail scene. Because the detail scene should always display the same configuration of data, you used a static-content based table to lay out three table cells—one cell for each bird sighting detail.

Finally, you implemented the `prepareForSegue` method in the master view controller, which allowed you to pass to the detail scene the `BirdSighting` object that's associated with the user's selection in the master list.

At this point in the tutorial, the code in `BirdsDetailViewController.h` should look like this:

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

@class BirdSighting;
```



```
@interface BirdsDetailViewController : UITableViewController

@property (strong, nonatomic) BirdSighting *sighting;
@property (weak, nonatomic) IBOutlet UILabel *birdNameLabel;
@property (weak, nonatomic) IBOutlet UILabel *locationLabel;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;
@end
```

The code in `BirdsDetailViewController.m` should look similar to this (not including template-provided code that you don't edit in this tutorial):

```
#import "BirdsDetailViewController.h"
#import "BirdSighting.h"
@interface BirdsDetailViewController ()
- (void)configureView;
@end

@implementation BirdsDetailViewController

#pragma mark - Managing the detail item

- (void)setSighting:(BirdSighting *) newSighting
{
    if (_sighting != newSighting) {
        _sighting = newSighting;

        // Update the view.
        [self configureView];
    }
}

- (void)configureView
{
    // Update the user interface for the detail item.
    BirdSighting *theSighting = self.sighting;
```



```
    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
    if (theSighting) {
        self.birdNameLabel.text = theSighting.name;
        self.locationLabel.text = theSighting.location;
       self.dateLabel.text = [formatter stringFromDate:(NSDate *)theSighting.date];
    }
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    [self configureView];
}
@end
```

The `prepareForSegue` method in `BirdsMasterViewController.m` should look similar to this (edits that you made to this file earlier in the tutorial are not shown):

```
#import "BirdsDetailViewController.h"
...
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender

{
    if ([[segue identifier] isEqualToString:@"ShowSightingDetails"]) {
        BirdsDetailViewController *detailViewController = [segue
destinationViewController];

        detailViewController.sighting = [self.dataController
objectInListAtIndex:[self.tableView indexPathForSelectedRow].row];
```



```
    }
}
...
@end
```

Enabling the Addition of New Items

At this point in the tutorial, the BirdWatching app displays a placeholder item in the master scene and additional information about the item in the detail scene. You've learned how easy it is to design and implement this common app design using storyboards.

In this chapter, you create a new scene that lets users enter information about a new bird sighting and add it to the master list. You'll learn how to:

- Add a new navigation controller hierarchy to the storyboard
- Present a scene modally
- Use Auto Layout to specify the visual relationships among UI elements
- Allow a destination scene to send information back to its source scene
- Improve the accessibility of your app

Create the Files for a New Scene

In this step, you create the interface and implementation files for the view controller that manages the add scene that lets users add a new bird-sighting to the master list. Then, you design the add scene on the canvas. You don't do much implementation of the view controller in this step, but it's useful to add the files to your project before you work with the scene on the canvas because doing so allows you to set up property declarations and action methods by dragging from the canvas to the appropriate class file.

To create the add scene view controller class files

1. Choose File > New > File (or press Command-N).
2. In the dialog that appears, select Cocoa Touch in the iOS section on the left.
3. In the main area of the dialog, select Objective-C Class and then click Next.
4. In the next pane of the dialog, enter `AddSightingViewController` in the Class field and choose `UITableViewController` in the Subclass pop-up menu.

   You want the add scene to be a table view controller subclass because each input text field is displayed in a table cell.

5. Make sure that the "Targeted for iPad" and "With XIB for user interface" options are not selected and then click Next.
6. In the next dialog that appears, choose the BirdWatching folder in the Group pop-up menu and click Create.

Now create the add scene in the storyboard file.

To create the add scene on the canvas

1. Select `MainStoryboard.storyboard` in the project navigator.

   To give yourself more room, close the document outline pane by clicking the button in the lower-left corner of the canvas that looks like this: 

   If you want, you can also adjust the canvas zoom level.

2. Drag a table view controller from the object library to the canvas.

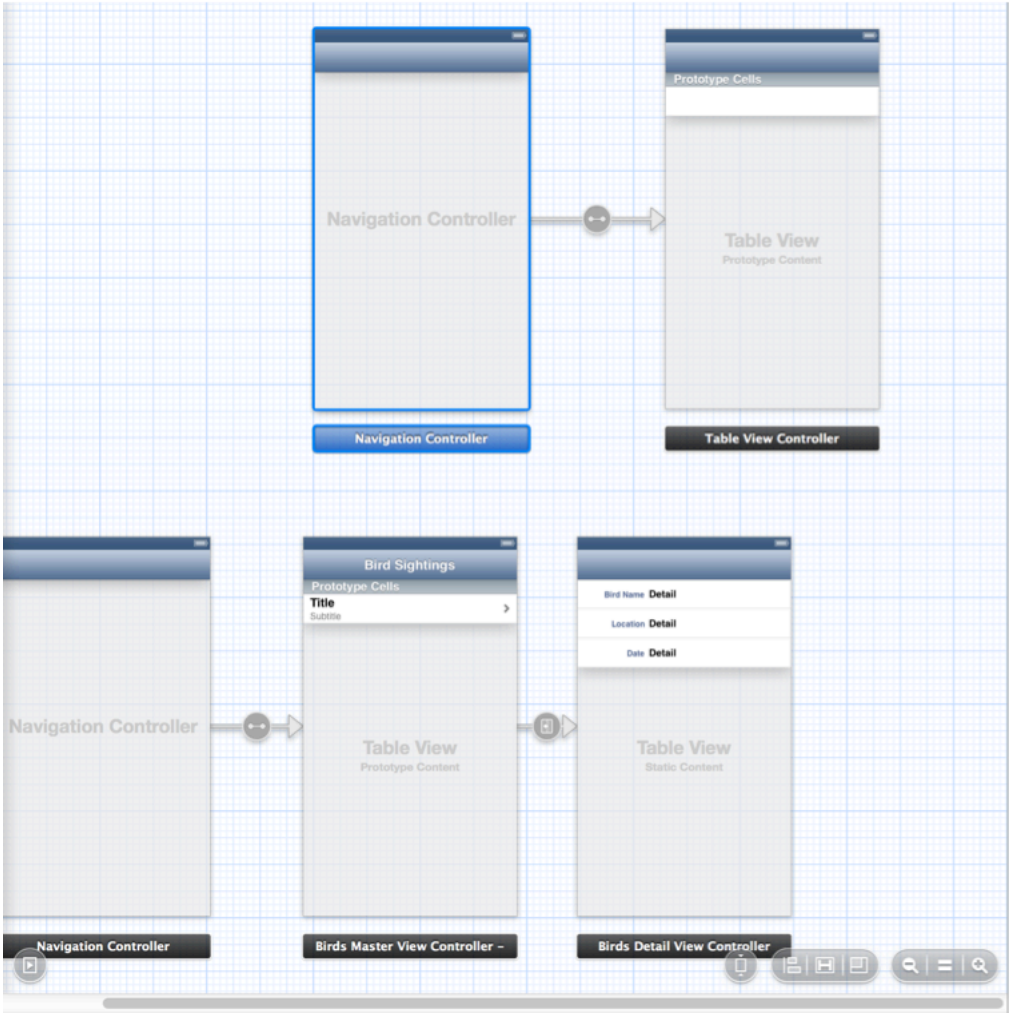
The add scene needs to include a Done button and a Cancel button so that users can either save their input or cancel the add action. Typically, Done and Cancel buttons belong in a navigation bar at the top of a screen. Although the add scene doesn't need to be part of a navigation hierarchy (because it doesn't allow users to transition to additional scenes), the easiest way to give it a navigation bar is to embed the add scene in a navigation controller. Embedding a scene in a navigation controller also has the following advantages:

- It ensures that the navigation bar doesn't scroll off the screen if the main content view scrolls. This is not a problem in this tutorial (because the add scene's table view has only two cells), but it's a good technique to know.
- It makes it easy to extend the current design to transition to additional scenes from the add scene. See "Next Steps" (page 87) for some ways in which you can enhance the BirdWatching app.

To embed the add scene in a navigation controller

1. On the canvas, click the status bar of the add scene to select it.
2. Choose Editor > Embed In > Navigation Controller.

Xcode adds a navigation controller scene to the canvas and automatically connects it to the add scene with a relationship. You should see something like this:



After you finish embedding the new table view controller in a navigation controller, Xcode warns you about a few issues. Specifically:

- The implementation of `AddSightingViewController` is potentially incomplete.
- Prototype table cells need reuse identifiers.
- The scene you just added to the canvas is currently unreachable.

The first issue occurs because Xcode automatically includes stubs for the table view data source methods in a new `UITableViewController` implementation file. (You implemented some of these methods for the master view controller in "To implement the table view data source methods in the master view controller" (page 35).) But unlike the table in the master scene, which must be able to display as many bird sighting items as

the user wants to enter, the add scene's table always displays the same configuration of content. Therefore, you can base the add scene's table on static content, as you did for the detail scene's table, rather than on dynamic prototypes, as you did for the master scene's table. When you base a table on static content, the table view controller automatically handles the data source needs of the table view, so you don't have to implement the data source methods.

The second issue arises because when you drag a new table view controller scene to the canvas, it is based on prototype cells by default. In the next section, when you change the add scene's table to be based on static content, you also resolve this issue, because static cells don't need reuse identifiers.

The third issue is that the add scene is not connected to any other scene in your app. You'll fix that situation later when you create a segue from the master scene to the add scene.

Because the add scene doesn't need the table view data source methods, you can delete (or comment out) the following stubs in `AddSightingViewController.m`:

- `numberOfSectionsInTableView`
- `numberOfRowsInSection`
- `cellForRowAtIndexPath`
- `didSelectRowAtIndexPath`

> **Note:** Xcode did not display similar warnings when you used static cells to design the detail scene's table, because there were no data source method stubs in the template-provided class files for the detail scene. There were no stubs because the template provides a default detail scene that is based on `UIViewController`, not `UITableViewController`.

Before you continue, you can also delete or comment out the default `initWithStyle` method because you will specify the table view's style on the canvas.

Design the UI of the Add Scene

Before you lay out the add scene, first specify the custom view controller class that manages it. If you don't do this, none of the work you do in this chapter will have any effect on the running app.

To specify the identity of the add scene

1. Select `MainStoryboard.storyboard` in the project navigator.
2. On the canvas, select the add scene—that is, the table view controller object you dragged out of the object library.

Make sure you select the add scene itself, and not the navigation controller in which you embedded it.

3. In the Identity inspector, choose `AddSightingViewController` in the Class pop-up menu.

To give users the ability to save information about a new bird sighting—or to abandon their input—you need to put Cancel and Done buttons in the add scene's navigation bar.

To add Cancel and Done buttons to the add scene

1. On the canvas, zoom in on the add scene.
2. One at a time, drag two bar button items from the object library, placing one item in each end of the add scene's navigation bar.

   You don't have to be very careful where you release the bar button items. As long as you drag the button to one side or the other of the navigation bar's center, Xcode automatically places it in the proper location.
3. Select the left bar button and ensure that the Attributes inspector is open.
4. In the Bar Button Item section of the Attributes inspector, choose Cancel in the Identifier pop-up menu.
5. Select the right bar button and choose Done in the Identifier pop-up menu.

As you did in the detail scene, you want to design static cells for the add scene's table view because the table view should always use the same layout to display the labels and input text fields. Before you can do this, you must change the content type of the table view from dynamic prototype to static cell.

To change the content type of the add scene's table

1. Select the add scene's table view, and open the Attributes inspector.
2. In the Attributes inspector, choose Static Cells in the Content pop-up menu.

The add scene needs only two table cells—one for the bird name and one for the sighting location—because the app uses the current date for the date of the sighting. Each cell needs a label and a text input field so that users know what information to enter. In this step, you lay out the top table cell first and then duplicate it for the bottom cell.

As you lay out the table cell, take advantage of Auto Layout to ensure that the UI elements behave correctly when the device rotates. Using the Auto Layout system, you define and prioritize layout **constraints**—which represent relationships between UI elements—and allow the system to lay out the UI in the way that best satisfies those constraints. (Auto Layout also helps you prepare for localization; to learn how to localize an app, see *Internationalize Your App* , which is part of *Start Developing iOS Apps Today (Retired)* .)

In the add scene, you want to allow the label to expand to fit its contents but you don't want it to cause the text field to become too narrow. Also, you want the space between the label and text field to remain constant. You define these layout relationships by giving a high priority to the label's Horizontal Content Hugging constraint, adding a minimum width constraint to the text field, and specifying a horizontal spacing constraint for the space between the two elements.

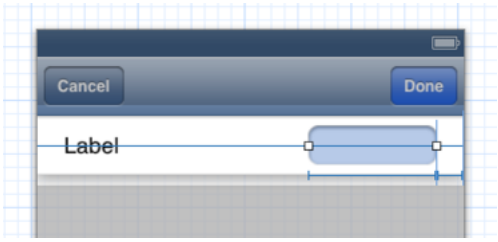
To lay out the first cell in the add scene's table

1. On the canvas, select two of the three default cells in the add scene's table view and press Delete.
2. Drag a label from the object library to the remaining table cell.
3. Keeping the label vertically centered in the cell, move the label to the left until you see a vertical dashed blue line appear at its left edge:

   As you can see above, Xcode displays a horizontal dashed blue line through the middle of a UI element when it's vertically centered in its parent view.

4. Drag a text field from the object library to the cell, moving it to the right—again, keeping it vertically centered—until you see a vertical dashed blue line appear at the field's right edge.

   After you add the label and the text field, you should see something like this:



   The solid horizontal blue line that runs through both the label and the text field is a Center Y Alignment Constraint, which ensures that both UI elements are vertically centered in their parent view (that is, in the table cell).

5. Click the Show Document Outline button in the bottom-left corner of the canvas to reveal the document outline.

6. In the Add Sighting View Controller Scene section of the document outline, select the label to reveal its current bounds and constraints.

   You should see something like this:

   The horizontal blue I-beam to the left of the label is a Horizontal Space Constraint, which specifies a system-defined space between the leading edge of the label and the left edge of the cell. (The text field has a similar Horizontal Space Constraint between its right—or trailing—edge, but you don't see it unless you select the text field.)

7. With the label still selected, open the Size inspector by clicking the Size button in the utility area.

   The Size inspector button looks like this: 

8. In the Content Hugging Priority section of the Size inspector, type 999 in the text field to the right of the Horizontal slider.

   A value of 1000 for a constraint priority means that the constraint is required. By giving the label a Horizontal Content Hugging Priority of 999, you specify that it's very important—but not required—that the label will expand in width to fit its contents.

9. On the canvas, select the text field.

10. In the Constraints section of the Size inspector, click the gear icon in the Width Equals section and choose Select and Edit.

    The Size inspector closes and the Attributes inspector opens, which displays information about the width constraint.

11. In the Width Constraint section of the Attributes inspector, choose Greater Than or Equal in the Relation pop-up menu.

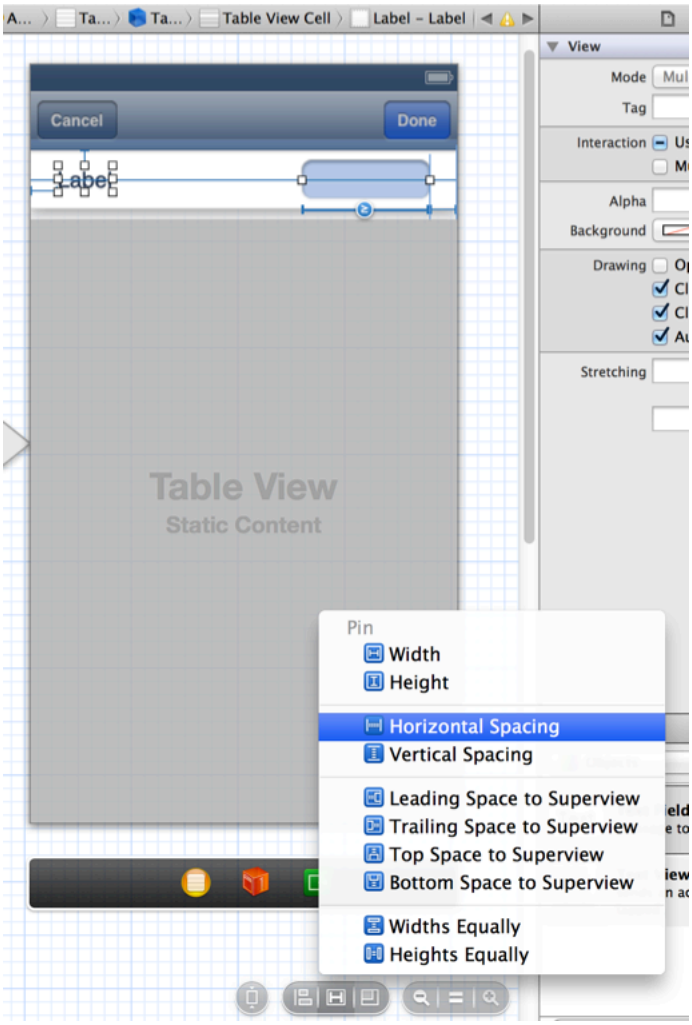
    Specifying Greater Than or Equal means that the text field will not get narrower than the default width of 97 points, no matter how wide the label becomes.

12. On the canvas, select both the label and the text field and click the pin constraint button in the lower-right corner of the canvas.

The pin constraint button is the middle button in the group that looks like this:

13. In the Pin menu that appears, choose Horizontal Spacing.

    You should see something like this:



14. In the Horizontal Space Constraint section of the Attributes inspector, select the Standard checkbox.

    Horizontal spacing refers to the space between two selected objects. Choosing the standard space between the label and the text field—and leaving the priority set to 1000—ensures that content in these two elements won't collide.

On the canvas, you should see something like this:

15. Select the text field and make the following choices in the Attributes inspector:
    - In the Capitalization pop-up menu, choose Words.
    - Ensure that the Keyboard pop-up menu is set to Default.
    - In the Return Key pop-up menu, choose Done.

After you finish laying out the first table cell, you can duplicate it to create the second cell. Then, make a few minor changes to the labels in both cells to customize them.

To create a second table cell and customize both cells

1. On the canvas, select the cell you created and press Command-D to duplicate it.

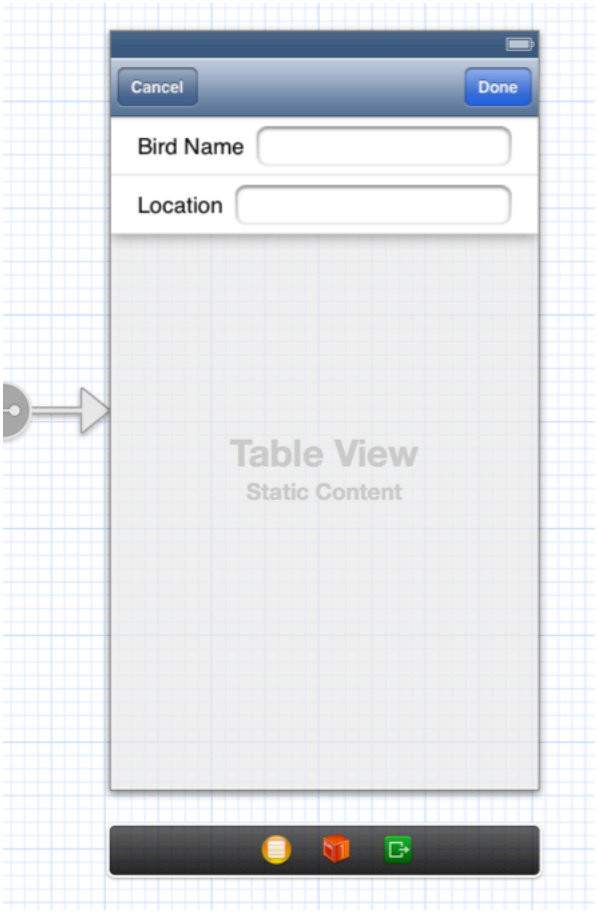
   The easiest way to select the cell—and not the label or the text field—is to click in one of the cell corners. The duplicated cell appears below the original cell.

2. Double-click the default text in each label and replace it with custom text.

   For the label in the top cell, type `Bird Name`; for the label in the bottom cell, type `Location`.

   Because Bird Name is longer than Location, the text field in the top cell becomes narrower than the text field in the bottom cell. You specified this behavior by requiring that the horizontal space between a label and a text field remain constant and by requiring the label to expand to fit its contents.

After you finish laying out the table cells in the add scene, the canvas should look something like this:



Note: If you've zoomed out from the canvas, you see the title of the add scene's view controller (that is, Add Sighting View Controller) in the scene dock instead of the proxy objects you can see above.

Connect the Master Scene to the Add Scene

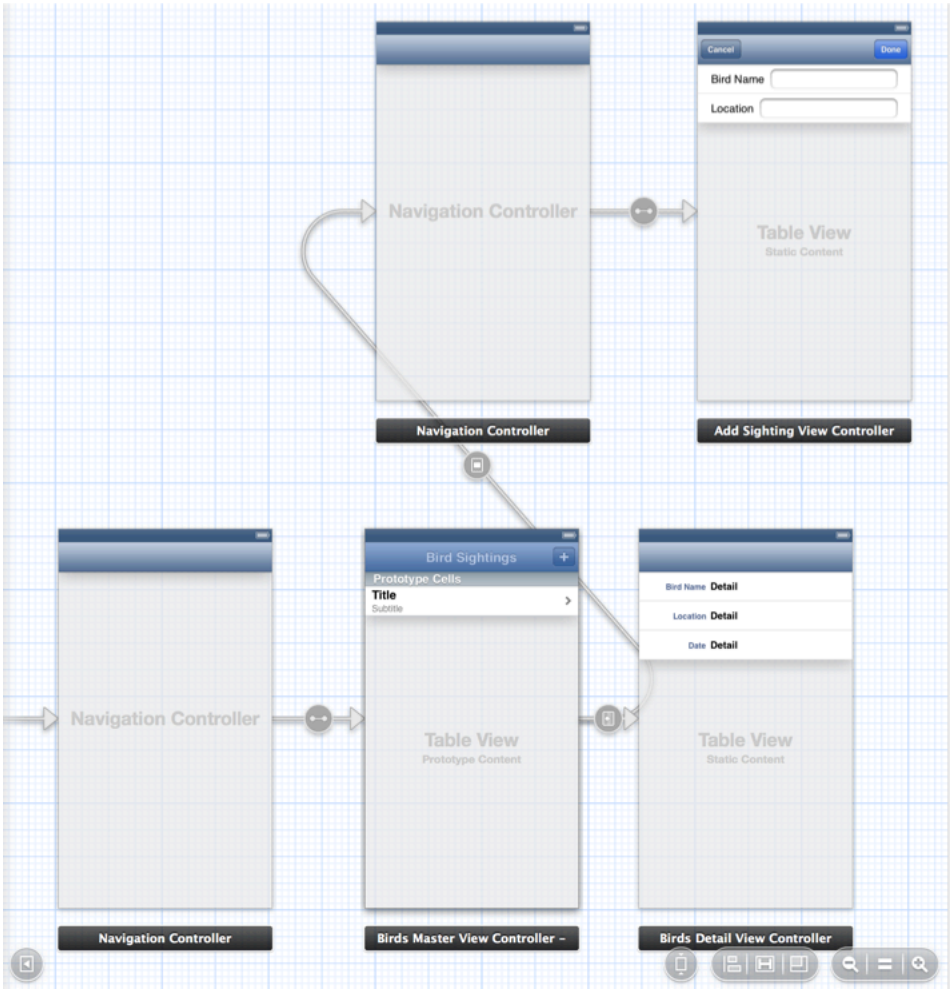
In iOS apps, users typically enter new information in a self-contained, modally presented screen. For example, when users tap the Add button in Contacts on iPhone, a new screen slides up in which they can enter information. When they finish the task (or they cancel it), the screen slides back down, revealing the previous screen.

To follow this design pattern, you want the add scene to slide up when the user taps an Add button in the master scene and slide back down when the user taps the add scene's Done or Cancel button. You enable the modal display of the add scene by creating a modal segue from the master scene to the add scene (you'll enable the dismissal of the add scene in a later step).

To allow the master scene to transition to the add scene

1. On the canvas, zoom in on the master scene.
2. Drag a bar button item out of the object library into the right end of the master scene's navigation bar.
3. With the bar button still selected, open the Attributes inspector and choose Add in the Identifier pop-up menu.
4. On the canvas, Control-drag from the Add button to the navigation controller in which you embedded the add scene.
5. In the Action Segue menu that appears, select modal.

After you create the modal segue, the canvas should look something like this:



Even though Xcode shows the new segue emerging from the middle of the master scene, and not from the Add button, the actual connection is associated with the button.

Recall that in "To send setup information to the detail scene" (page 49), you implemented the `prepareForSegue` method so that the master scene sends a `BirdSighting` object to the detail scene when the user selects an item in the master list. But when the master scene prepares to segue to the add scene, it does not need to send any data for the add scene to display, so you don't need to create an ID for the add-scene segue or update the master scene's `prepareForSegue` method.

Prepare to Handle Text Field Input

The add scene's view controller needs access to the two text fields you added in "Connect the Master Scene to the Add Scene" (page 63) so that it can capture the information that users enter. You also want the view controller to respond appropriately when the user taps the Cancel and Done buttons. Because you already added the `AddSightingViewController` code files to the project, you can let Xcode create the appropriate code as you make connections between the text fields on the canvas and the view controller header file. (You'll be writing the code to handle the buttons yourself.)

Each text field in the add scene needs an outlet in the view controller code so that the view controller and the text field can communicate at runtime. Set up the outlets for the input text fields by Control-dragging from each element to the view controller header file.

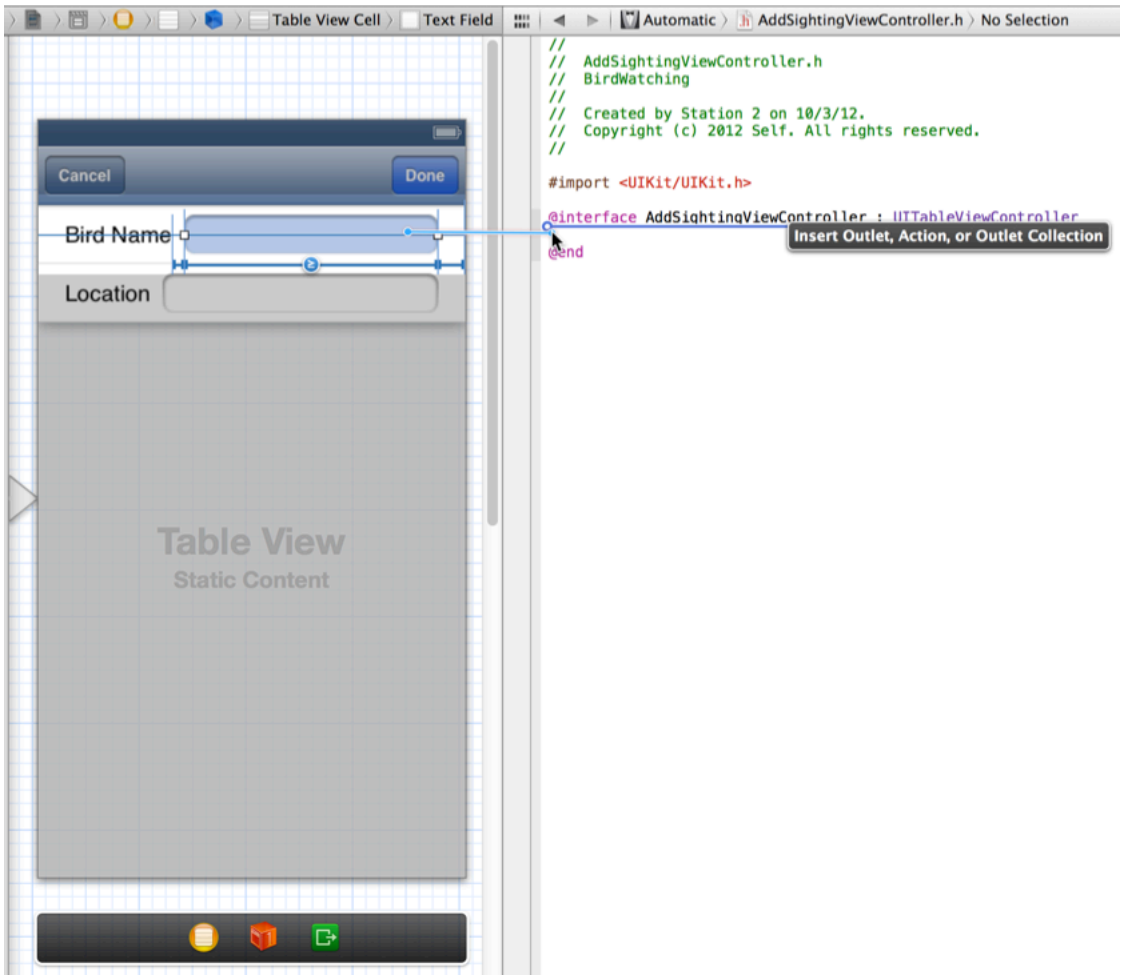
To create outlets for the input text fields

1. On the canvas, double-click the add scene to zoom in on it and select it.
2. In the Xcode toolbar, click the Utility button to hide the utility area and click the Assistant editor button to display the Assistant editor pane.

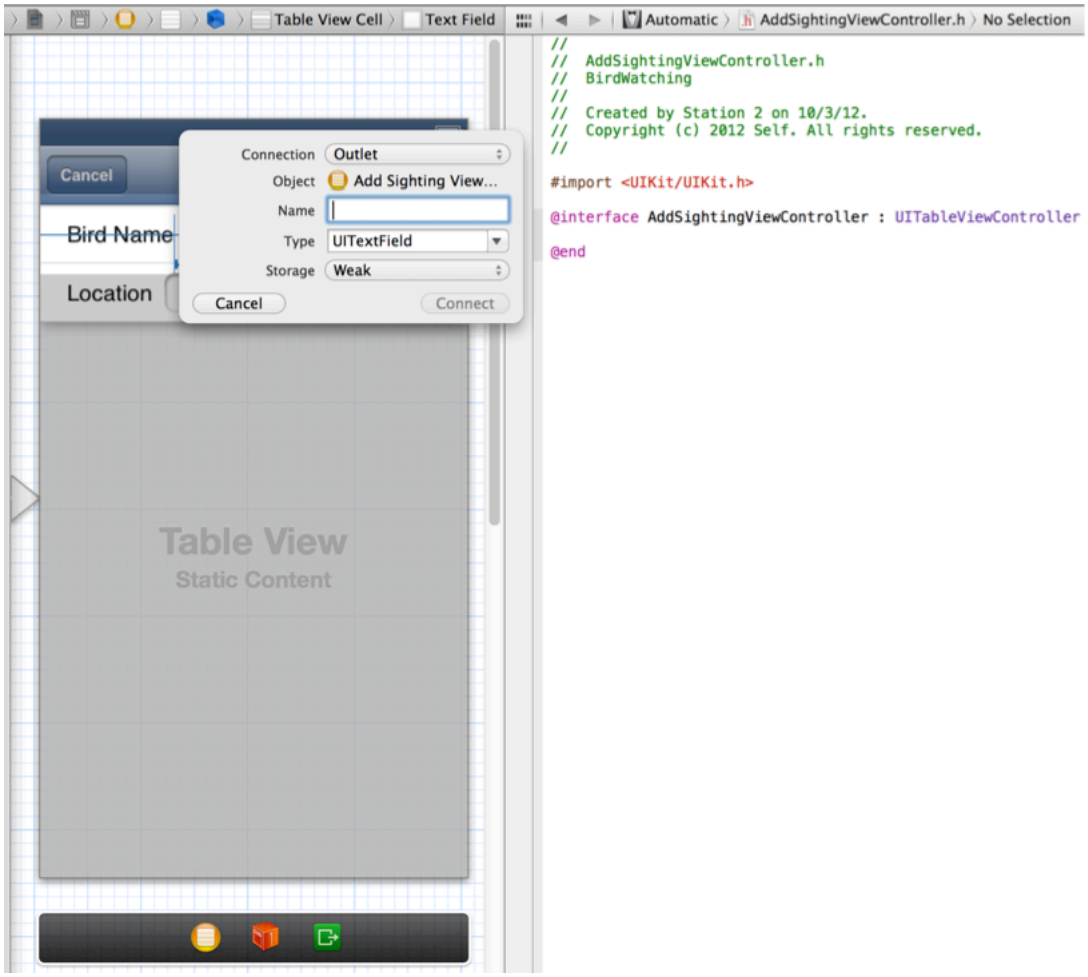
   The Assistant editor button is to the left of the three View buttons (which include the Utility button) and it looks like this: 
3. Make sure that the Assistant displays the view controller's header file (that is, `AddSightingViewController.h`).

   If the Assistant displays a different file, it might be because you forgot to set the identity of the add scene to `AddSightingViewController` or because the add scene isn't selected on the canvas.
4. Control-drag from the text field in the Bird Name cell to the method declaration area in the header file.

As you Control-drag, you should see something like this:



When you release the Control-drag, Xcode displays a popover in which you configure the outlet connection you just made:



5. In the popover that appears when you release the Control-drag, configure the outlet in the following way:
   - Make sure that the Connection pop-up menu contains Outlet.
   - In the Name field, type `birdNameInput`.
   - Make sure that the Type field contains `UITextField` and that the Storage pop-up menu contains `Weak`.

6. In the popover, click Connect.
7. Perform actions similar to steps 3, 4, and 5 to create an outlet for the text field in the Location table cell:

- Control-drag from the text field to an area between the `@property` statement Xcode added for `birdNameInput` and the `@end` statement in the header file.
- In the popover that appears when you release the Control-drag, make sure that the Connection menu contains Outlet and type `locationInput` in the Name field.
- Make sure that the Type field contains `UITextField` and that the Storage menu contains `Weak`; then click Connect.

By Control-dragging to establish an outlet connection between each text field and the view controller, you told Xcode to add code to the view controller files. Specifically, Xcode added the following declarations to `AddSightingViewController.h`:

```
@property (weak, nonatomic) IBOutlet UITextField *birdNameInput;
@property (weak, nonatomic) IBOutlet UITextField *locationInput;
```

Xcode also automatically synthesizes the appropriate accessor methods for these properties, although no code is added to `AddSightingViewController.m`.

Now that you've set up the outlets, you need to manage the interaction between the text fields and the system keyboard. To give yourself more room, click the Standard editor button to close the Assistant editor and expand the canvas editing area. The Standard editor button looks like this: 

As you learned in *Your First iOS App* (which is part of *Start Developing iOS Apps Today (Retired)*), an app can make the system keyboard appear and disappear as a side effect of toggling the first responder status of a text entry element, such as a text field. The `UITextFieldDelegate` protocol includes the `textFieldShouldReturn:` method, which a text field calls when the user wants to dismiss the keyboard. To revoke the text field's first responder status—and cause the keyboard to dismiss—you choose an app object to act as the text field's delegate and implement `textFieldShouldReturn:`. In the BirdWatching app, set each text field's delegate to the add scene and implement the required protocol method in `AddSightingViewController.m`.

To set the delegate for the text fields and implement the textFieldShouldReturn: method

1. On the canvas, double-click the add scene to zoom in on it, if necessary.
2. Click the top text field to select it.
3. Control-drag from the text field to the view controller proxy object displayed in the scene dock.

   The view controller proxy object is represented by the yellow sphere.
4. In the Outlets section of the translucent panel that appears, select `delegate`.

5. In the add scene, select the bottom text field and perform steps 3 and 4 again.
6. In the project navigator, select `AddSightingViewController.h`.
7. Update the `@interface` code line to look like this:

```
@interface AddSightingViewController : UITableViewController
<UITextFieldDelegate>
```

   Adding `UITextFieldDelegate` to the `@interface` line in this way signals that the add scene adopts the text field delegate protocol.

8. In the project navigator, select `AddSightingViewController.m`.
9. Implement the `UITextFieldDelegate` protocol method by adding the following code to the implementation block:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
if ((textField == self.birdNameInput) || (textField ==
self.locationInput)) {
        [textField resignFirstResponder];
}
    return YES;
}
```

   The `if` statement in this implementation ensures that the text field resigns its first responder status regardless of which button the user taps.

After the user finishes entering a new bird sighting, the add scene needs to package the information and pass it to the master scene for addition to the list. You already created a class whose instances represent bird sightings, so now you need to give the add sighting view controller the ability to create new `BirdSighting` objects and send them back to the master view controller.

To give the add scene access to BirdSighting objects

1. In the document outline, select `AddSightingViewController.h`.
2. Add a forward declaration of the bird-sighting class so that the add sighting view controller recognizes its type.

   Add the following line of code after the `#import` statement:


```
@class BirdSighting;
```

3. Declare a bird-sighting property.

   Before the `@end` statement, add the following code:

```
@property (strong, nonatomic) BirdSighting *birdSighting;
```

Get the User's Input

When the user taps the Cancel or Done button in the add scene, you want to dismiss the scene and return to the master list, updated with the user's input. To enable this behavior, you use unwind segues and custom methods that handle the new bird-sighting information. Unlike a standard segue—which instantiates the destination scene before displaying it—an **unwind segue** allows the source scene to target a destination scene that already exists. (In both unwind and standard segues, the source scene can implement `prepareForSegue` to send information to the destination.)

To make itself available as the destination of an unwind segue, a scene must declare unwind action methods. In this tutorial, you want the add scene's Cancel and Done buttons to trigger unwind segues that transition to the master scene, so you declare and implement these methods in `BirdsMasterViewController`.

To set up unwind action methods for the Done and Cancel buttons

1. In the project navigator, select `BirdsMasterViewController.h`.
2. Declare the unwind action methods by adding the following code before the `@end` statement:

```
- (IBAction)done:(UIStoryboardSegue *)segue;
- (IBAction)cancel:(UIStoryboardSegue *)segue;
```

3. In the project navigator, select `BirdsMasterViewController.m`.
4. Import the `AddSightingViewController` header file.

   Add the following code line above the `@implementation` statement:

```
#import "AddSightingViewController.h"
```

5. Implement the `done` method by adding the following code to the implementation block:


```
- (IBAction)done:(UIStoryboardSegue *)segue
{
    if ([[segue identifier] isEqualToString:@"ReturnInput"]) {

        AddSightingViewController *addController = [segue
sourceViewController];
        if (addController.birdSighting) {
            [self.dataController
addBirdSightingWithSighting:addController.birdSighting];
            [[self tableView] reloadData];
        }
        [self dismissViewControllerAnimated:YES completion:NULL];
    }
}
```

Notice that the code above specifies a segue identifier—that is, `ReturnInput`—that's not yet associated with a segue on the canvas. In a later step, you enter this ID in the Attributes inspector for the unwind segue you create for the Done button.

6. Still in `BirdsMasterViewController.m`, implement the `cancel` method by adding the following code to the implementation block:

```
- (IBAction)cancel:(UIStoryboardSegue *)segue
{
    if ([[segue identifier] isEqualToString:@"CancelInput"]) {
        [self dismissViewControllerAnimated:YES completion:NULL];
    }
}
```

As with the segue ID specified in the `done` method, in a later step you enter `CancelInput` in the Attributes inspector for the unwind segue you create for the Cancel button.

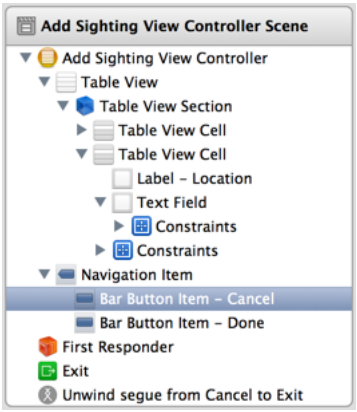
When you added the declarations of the `cancel` and `done` methods to the master view controller, you allowed it to advertise itself as a destination for unwind segues. Now you can create unwind segues for the Cancel and Done buttons and associate them with these methods.

To set up unwind segues for the Cancel and Done buttons

1. Select `MainStoryboard.storyboard` in the project navigator.
2. If necessary, zoom in and adjust the canvas so that you can focus on the add scene.
3. Control-drag from the Cancel button in the add scene to the unwind segue's destination proxy object in the scene dock.

   The destination proxy object is represented by the Exit icon, which looks like this: 
4. In the Action Segue menu that appears when you release the Control-drag, choose the `cancel:` action method.

   After you choose `cancel:`, Xcode displays the segue in the Add Sighting View Controller Scene section of the document outline:



5. In the same way, create an unwind segue for the Done button.
   - Control-drag from the Done button in the add scene to the Exit icon in the scene dock.
   - In the Action Segue menu that appears when you release the Control-drag, choose `done:`.
6. In the Add Sighting View Controller Scene section of the document outline, select "Unwind segue from Cancel to Exit."
7. If necessary, click the Utilities button to open the utility area.
8. In the Storyboard Unwind Segue section of the Attributes inspector, enter `CancelInput` in the Identifier text field.

   As with other segues you've created, it's crucial to use the same ID value in both the Identifier text field and in code.

9. In the same way, identify the Done button's unwind segue by entering the ID you specified in the master scene's `done` method:
    - In the Add Sighting View Controller Scene section of the document outline, select "Unwind segue from Done to Exit."
    - In the Storyboard Unwind Segue section of the Attributes inspector, enter `ReturnInput` in the Identifier text field.

Even though the master scene is prepared to handle the unwind actions in its `cancel` and `done` methods, it doesn't yet have access to the user's input. To fix this situation, implement `prepareForSegue` in the add scene to create a new `BirdSighting` object from the user's input and send it to the master scene.

To send the user's input to the master scene

1. In the project navigator, select `AddSightingViewController.m`.
2. Import `BirdSighting.h` by adding the following code line before the `@interface` statement:

```
#import "BirdSighting.h"
```

3. Implement the `prepareForSegue` method by adding the following code to the implementation block:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"ReturnInput"]) {
        if ([self.birdNameInput.text length] || [self.locationInput.text
 length]) {
            BirdSighting *sighting;
            NSDate *today = [NSDate date];
            sighting = [[BirdSighting alloc]
initWithName:self.birdNameInput.text location:self.locationInput.text
date:today];
            self.birdSighting = sighting;
        }
    }
}
```


Enhance the Accessibility of Your App

An accessible app reports information about its UI elements and behavior so that people with disabilities can use an assistive technology—such as VoiceOver—to interact with the app. Every app should be accessible.

Standard UI elements are accessible by default because they automatically report information to VoiceOver. VoiceOver uses this information—provided in an element's *accessibility attributes* —to describe the element and its current state to VoiceOver users.

Although the BirdWatching app is also accessible by default—because it contains only standard UI elements—it's important to look for ways to enhance the experience for VoiceOver users. In this tutorial, you'll improve the accessibility of your app by describing what the master scene's Add button does.

Before you make any changes, it's a good idea to find out what VoiceOver already knows about your app. To help you do this, iOS Simulator provides Accessibility Inspector.

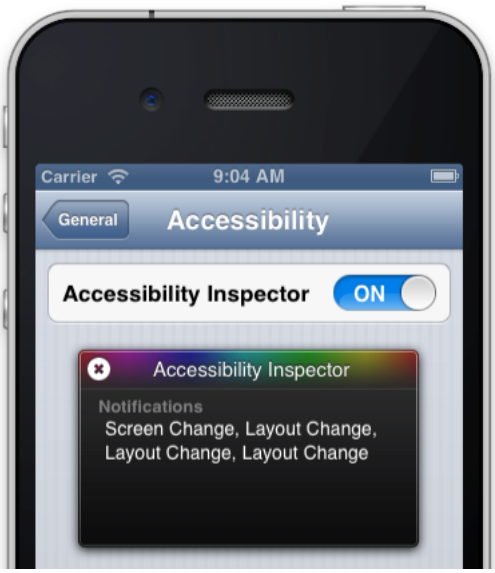
To test the accessibility of your app

1. In Xcode, click Run to open your app in iOS Simulator.
2. At the bottom of the simulated device screen, click the Home button.
3. Open the Settings app.

   You might have to navigate to a different Home screen to find the Settings app.

4. In Settings > General > Accessibility, turn on Accessibility Inspector.

   The Accessibility Inspector panel—which floats above all other content in iOS Simulator—appears onscreen. When Accessibility Inspector is active, the panel displays accessibility information about an element, which looks something like this:



When Accessibility Inspector is active, you must perform the following actions to interact with an onscreen element:

a. Single-click the element to focus on it.

   When Accessibility Inspector focuses on an element, it draws a shaded box around the element.

b. Triple-click the focused element to activate it.

   For example, instead of single-clicking in a text field to bring up the keyboard, you single-click the text field to focus on it and then triple-click in it.

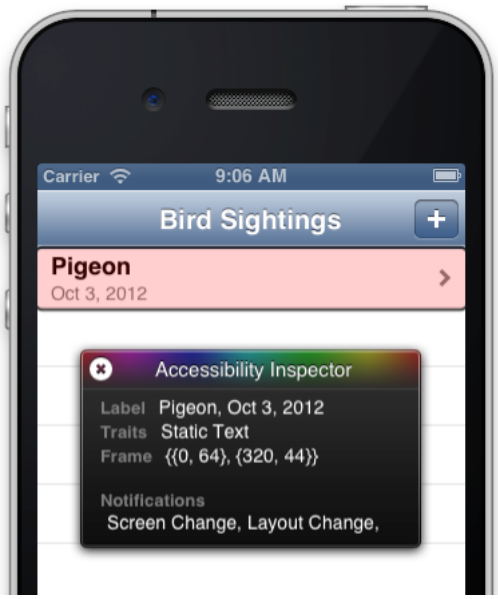
5. Click the close button in the upper-left corner of the Accessibility Inspector panel to deactivate it.

   When Accessibility Inspector is inactive, it looks like this:

   When you deactivate Accessibility Inspector it means that you don't have to use the focus and triple-click technique to activate UI elements in iOS Simulator. You can reactivate Accessibility Inspector when you want to view an element's accessibility information.

6. Click the Home button to close Settings and return to the Home screen.
7. On the Simulator Home screen, locate and open the BirdWatching app.
8. Click the close button in the Accessibility Inspector panel to reactivate it.
9. In BirdWatching, click the first table row to view its accessibility information.

You should see something like this:



Notice that the table cell automatically combines the title and subtitle text into a single accessibility label attribute that VoiceOver speaks to users.

10. Click the Add button.

    As you can see here, the Add button's default label is accurate, but not particularly helpful:

11. Deactivate Accessibility Inspector and stop iOS Simulator.

The Stop button is in the left end of the Xcode toolbar, and it looks like this: 

Whether you stop or quit iOS Simulator, Accessibility Inspector remains on until you turn it off.

In "To allow the master scene to transition to the add scene" (page 64), you used system-provided items to create a standard Add button, so VoiceOver already knows that the element is a button and that the plus sigh (+) means add. But VoiceOver can't tell users about the type of thing that gets added, because the button doesn't supply any other information. To clarify the Add button's behavior, you can supply a **hint**, which is an accessibility attribute that describes the results of interacting with a control.

The hint attribute is represented by a property that's available to objects that implement the `UIAccessibility` protocol. Because standard UI elements—such as buttons, text fields, and sliders—implement the `UIAccessibility` protocol by default, you can set this property in your code.

To provide an accessibility hint for the Add button

1. In the project navigator, select `BirdsMasterViewController.m`.
2. In the `viewDidLoad` method, set the `accessibilityHint` property.

   Add the following line of code after the `[super viewDidLoad];` statement:

```
self.navigationItem.rightBarButtonItem.accessibilityHint = @"Adds a new
 bird-sighting event";
```


Run the app again and activate Accessibility Inspector. When you click the Add button this time, you should see something like this:

Before you do additional testing, turn off Accessibility Inspector in iOS Simulator.

To turn off Accessibility Inspector

1. If necessary, click the close button in the Accessibility Inspector panel to deactivate it.
2. Click the Home button and open Settings.
3. Choose General > Accessibility and turn off Accessibility Inspector.

Test the App

At this point in the tutorial, Xcode should not report any errors. If it does, check the code that you added in this chapter against the code listings in "Recap" (page 81).

Run the app again. When BirdWatching opens in iOS Simulator, there are several things you should test.

Try adding new bird sighting information to the master list. Make sure that you can select the new bird in the master scene and see details about it in the detail scene.

Change the device orientation to landscape while you're in the add scene. Each text field should automatically expand in width, but it should remain separated from its label and the right edge of the screen by constant amounts.

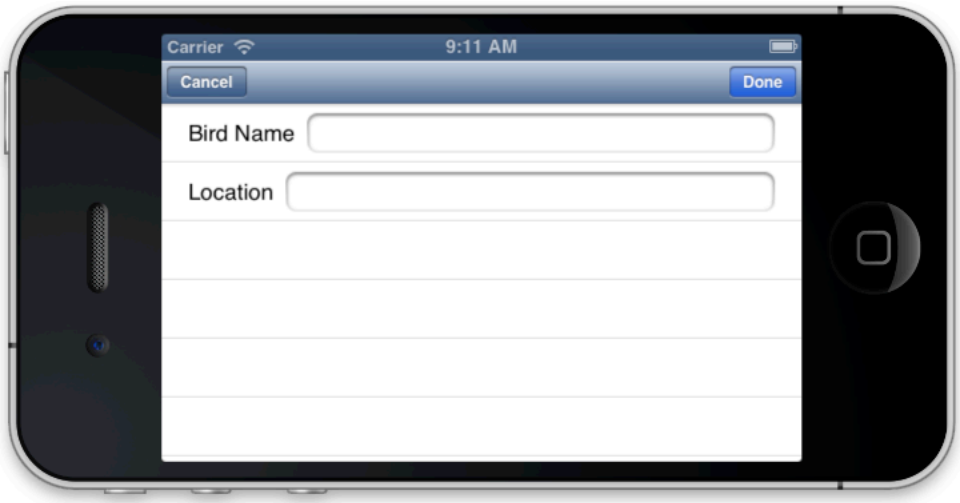
To change the device orientation in iOS Simulator

Do one of the following:

- Choose Hardware > Rotate Left (or press Command-Left Arrow).
- Choose Hardware > Rotate Right (or press Command-Right Arrow).

Each action performs a 90° rotation in the direction you specify.

If you choose Rotate Left, for example, you should see something like this:



Change the simulated device to iPhone 5. The master and detail scenes should display additional table rows automatically. In the add scene, change the device orientation to landscape and confirm that the text fields still behave as designed.

To change the simulated device in iOS Simulator

- Choose Hardware > Device > iPhone (Retina 4-inch).

Congratulations! You've created an app that enables some the most common iOS user experiences, such as navigating through a stack of progressively more detailed screens and entering new information in a modal screen. If you're having any difficulties getting the app to run correctly, be sure to browse the advice in "Troubleshooting" (page 85) and check your code against the complete listings in "Code Listings" (page 90). And if you're ready for some challenges, read "Next Steps" (page 87), which suggests some ways in which you can expand and improve the BirdWatching app.

Recap

In this chapter, you created the class files for a new scene managed by a table view controller. When you created this scene on the canvas, you designed a static-content–based table that contains one cell for each of the two pieces of information that users can enter about a new bird sighting. In doing this, you learned that a table view controller that manages a static content–based table automatically takes care of the table's data source needs. You also embedded the new scene in a navigation controller so that it can display the Cancel and Done buttons in a navigation bar.

Next, you took advantage of Xcode's code-addition features to create stubs for action methods by Control-dragging from items on the canvas to the `AddSightingViewController` header file. You also specified that `AddSightingViewController` should adopt the text field delegate protocol so that—by implementing the `textFieldShouldReturn` method—the keyboard disappears when users finish entering text.

You also learned how to set up unwind actions in the master view controller so that the buttons in the add scene can trigger unwind segues and pass information back to the master scene. Finally, you learned how to test the accessibility of your app and enhance the experience for VoiceOver users by updating an accessibility attribute.

The code in the new `AddSightingViewController.h` file should look like this:

```
#import <UIKit/UIKit.h>
@class BirdSighting;
@interface AddSightingViewController : UITableViewController <UITextFieldDelegate>
@property (weak, nonatomic) IBOutlet UITextField *birdNameInput;
@property (weak, nonatomic) IBOutlet UITextField *locationInput;
@property (strong, nonatomic) BirdSighting *birdSighting;
@end
```

The code in the new `AddSightingViewController.m` file should look like this:

```
#import "AddSightingViewController.h"
#import "BirdSighting.h"
@interface AddSightingViewController ()

@end

@implementation AddSightingViewController
```

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    if ((textField == self.birdNameInput) || (textField == self.locationInput)) {
        [textField resignFirstResponder];
    }
    return YES;
}
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Uncomment the following line to preserve selection between presentations.
    // self.clearsSelectionOnViewWillAppear = NO;

    // Uncomment the following line to display an Edit button in the navigation
bar for this view controller.
    // self.navigationItem.rightBarButtonItem = self.editButtonItem;
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"ReturnInput"]) {
        if ([self.birdNameInput.text length] || [self.locationInput.text length])
 {
            BirdSighting *sighting;
            NSDate *today = [NSDate date];
            sighting = [[BirdSighting alloc] initWithName:self.birdNameInput.text
 location:self.locationInput.text date:today];
            self.birdSighting = sighting;
        }
    }
```



```
}
@end
```

The updated `BirdsMasterViewController.h` file should look like this:

```
#import <UIKit/UIKit.h>
@class BirdSightingDataController;
@interface BirdsMasterViewController : UITableViewController
@property (strong, nonatomic) BirdSightingDataController *dataController;
- (IBAction)done:(UIStoryboardSegue *)segue;
- (IBAction)cancel:(UIStoryboardSegue *)segue;
@end
```

The updated `BirdsMasterViewController.m` file should include the following additional code lines (edits you made to this file earlier in the tutorial are not shown):

```
...
#import "AddSightingViewController.h"
...
@implementation BirdsMasterViewController
...
- (IBAction)done:(UIStoryboardSegue *)segue
{
    if ([[segue identifier] isEqualToString:@"ReturnInput"]) {

        AddSightingViewController *addController = [segue sourceViewController];
        if (addController.birdSighting) {
            [self.dataController
addBirdSightingWithSighting:addController.birdSighting];
            [[self tableView] reloadData];
        }
        [self dismissViewControllerAnimated:YES completion:NULL];
    }
}
- (IBAction)cancel:(UIStoryboardSegue *)segue
{
```



```
    if ([[segue identifier] isEqualToString:@"CancelInput"]) {
        [self dismissViewControllerAnimated:YES completion:NULL];
    }
}
...
@end
```


Troubleshooting

If you have trouble getting the BirdWatching app to work correctly, try the problem-solving approaches described in this chapter.

Code and Compiler Warnings

If the app isn't working as it should, first compare your code with the code in "Code Listings" (page 90).

The tutorial code should compile without any warnings. If Xcode reports a warning, it's a good idea to treat the warning in the same way that you treat errors. Objective-C is a very flexible language, and sometimes a warning is the only indication you receive when there is an issue that might cause an error.

Storyboard Items and Connections

When you're accustomed to fixing all of an app's problems in code, it's easy to forget to check the objects in the storyboard. A great advantage of a storyboard is that, because it captures both the appearance and some of the configuration of app objects, you have less coding to do. To benefit from this advantage, it's important to examine the storyboard, as well as the code, when an app doesn't behave as you expect. Here are some examples.

A scene does not seem to receive the data you send to it in the prepareForSegue method. Check the segue's identifier in the Attributes inspector. If you forget to give the segue the same identifier that you use in the `prepareForSegue` method, you can still transition to the scene during testing, but the scene won't display the data that you send to it.

The same result occurs when you misspell a segue's identifier in the `prepareForSegue` method. Xcode does not warn you when a segue is missing its identifier, or when the `prepareForSegue` method uses an incorrect identifier, so it's important to check these values for yourself.

None of your changes to a custom view controller class seem to affect its scene. Check the scene's class in the Identity inspector. If the name of your custom view controller class is not displayed in the Class pop-up menu, Xcode does not apply your changes to the scene.

Nothing happens when you click the Cancel or Done buttons in the add scene's navigation bar. Make sure each button is connected to its unwind action. Control-click the button on the canvas or in the document outline and confirm that the connection in the Triggered Segues section is active and correct.

the text fields appear to work, but the data you enter is not displayed in the master list. Make sure the view controller's outlets are connected to the text fields.

Delegate Method Names

A common mistake related to using delegates is to misspell the delegate method name. Even if you've set the delegate object correctly, if the delegate doesn't use the right name in its method implementation, the correct method won't be invoked. It's usually best to copy and paste delegate method declarations—such as `textFieldShouldReturn:`—from a reliable source (such as the documentation or the declaration in a header file).

Next Steps

In this tutorial, you created a simple but functional app. To build on the knowledge you gained, consider extending the BirdWatching app in some of the ways described in this chapter.

Improve the User Interface and User Experience

To meet the high expectations of iOS users, an app must have a great user interface and user experience. Although it's usually best to avoid adding excessive decoration, the BirdWatching app might look better if it displayed coloring or even a subtle image in some of the view backgrounds.

iOS users generally expect to be able to use iOS-based devices in any orientation, so it's a good idea to support different orientations in the apps you develop. As you update the app's UI, be sure to define constraints that help UI elements remain properly positioned when the device is rotated.

Although the BirdWatching app makes it easy to add a new bird-sighting event, the user experience isn't ideal. For one thing, the Done button is active as soon as the add scene appears; it would be better if it became active after the user taps a key on the keyboard. In addition, the text fields don't provide a Clear button that allows users to quickly erase their input.

Add More Functionality

The best iOS apps include just the right amount of functionality to make it easy and enjoyable for users to accomplish the main task. The BirdWatching app makes it easy to perform the main task, but it could be more enjoyable to use. Here are a few improvements to consider:

- Instead of asking users to enter a bird name in a text field, display a list of bird names from which they can choose.
- Allow users to enter a specific date, rather than automatically using today's date.
- Ask users to enable Location Services so that the app can suggest the current location as the bird-sighting location. And instead of asking users to enter a location in a text field, display a map on which they can select a location.

- Let users edit and sort the master list. For some advice on how to enable these actions, see *Table View Programming Guide for iOS*. And recall that the Master-Detail template provides an Edit button (and some support for rearranging the table) by default. You might start on this task by removing the comment symbols around the appropriate code in `BirdsMasterViewController.m`.
- Allow users to add an image to a bird sighting. The app could let users choose from a set of stock images or from their own photos.
- Make the master list persistent so that users's bird sightings are displayed every time they restart the app.
- Adopt iCloud storage. When you support iCloud, changes that users make in one instance of an app are automatically propagated to their other devices, so that other instances of the app can see them, too. To get started learning how to enable iCloud in an app, see *Your Third iOS App: iCloud*.

Additional Improvements

As you learn more about iOS app development, consider making the following changes to the BirdWatching app:

- Support localization and accessibility. An app that is localized for different locales and accessible to users with disabilities enjoys a much wider customer base than an app without these features. Xcode and Auto Layout help make the internationalization and localization processes easy; to learn more, start by reading "Localized Resource Files" in *iOS App Programming Guide*. Use Accessibility Inspector in iOS Simulator to find places where your app can provide a better experience for VoiceOver users; to learn more, see "_Testing the Accessibility of Your iOS App".
- Optimize the code. High performance is critical to a good user experience on iOS. Learn to use the various performance tools provided with Xcode, such as Instruments, to tune your app so that it minimizes its resource requirements.
- Add unit tests. Testing ensures that if the implementation of a method changes, the method still works as advertised. You can either create a new version of the project that sets up unit testing from the start, or you can use the current project and choose File > New > New Target, select the Other category, and then select the template Cocoa Unit Testing Bundle. Examine the project to see what Xcode adds when you incorporate unit testing. To learn about unit testing, see *Xcode Unit Testing Guide*.
- Use Core Data to manage the model layer. Although it takes some work to learn how to use Core Data, it can help streamline the code required to support the model layer. Consider working through *Core Data Tutorial for iOS* to get started learning about this technology.
- Ensure that the app runs on a device. It's a good idea to familiarize yourself with the process of installing and testing an app on a device, because these tasks are prerequisites for submitting an app to the App Store.

- Make the app universal. iOS users often expect to be able to run their favorite apps on all types of iOS-based devices. Making an app universal can require additional work on your part; in particular, it generally requires that you create two different user interfaces, even if you reuse most of the same underlying code. To learn more about the steps you need to take to make an app universal, see “Creating a Universal App” in *iOS App Programming Guide*.

Code Listings

This appendix contains code listings for the interface and implementation files of the BirdWatching project. The listings do not include comments or methods that you do not edit.

Model Layer Files

This section contains listings for the following files:

- `BirdSighting.h`
- `BirdSighting.m`
- `BirdSightingDataController.h`
- `BirdSightingDataController.m`

BirdSighting.h

```
#import <Foundation/Foundation.h>


@interface BirdSighting : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *location;
@property (nonatomic, strong) NSDate *date;
-(id)initWithName:(NSString *)name location:(NSString *)location date:(NSDate
*)date;
@end
```

BirdSighting.m

```
#import "BirdSighting.h"


@implementation BirdSighting
```



```
-(id)initWithName:(NSString *)name location:(NSString *)location date:(NSDate
*)date
{
    self = [super init];
    if (self) {
        _name = name;
        _location = location;
        _date = date;
        return self;
    }
    return nil;
}
@end
```

BirdSightingDataController.h

```
#import <Foundation/Foundation.h>
@class BirdSighting;
@interface BirdSightingDataController : NSObject
@property (nonatomic, copy) NSMutableArray *masterBirdSightingList;
- (NSUInteger)countOfList;
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex;
- (void)addBirdSightingWithSighting:(BirdSighting *)sighting;
@end
```

BirdSightingDataController.m

```
#import "BirdSightingDataController.h"
#import "BirdSighting.h"
@interface BirdSightingDataController ()
- (void)initializeDefaultDataList;
@end
@implementation BirdSightingDataController
- (void)initializeDefaultDataList {
    NSMutableArray *sightingList = [[NSMutableArray alloc] init];
```



```
    self.masterBirdSightingList = sightingList;
    BirdSighting *sighting;
    NSDate *today = [NSDate date];
   sighting = [[BirdSighting alloc] initWithName:@"Pigeon" location:@"Everywhere"
 date:today];
    [self addBirdSightingWithSighting:sighting];
}
- (void)setMasterBirdSightingList:(NSMutableArray *)newList {
    if (_masterBirdSightingList != newList) {
        _masterBirdSightingList = [newList mutableCopy];
    }
}
- (id)init {
    if (self = [super init]) {
        [self initializeDefaultDataList];
        return self;
    }
    return nil;
}
- (NSUInteger)countOfList {
    return [self.masterBirdSightingList count];
}
- (BirdSighting *)objectInListAtIndex:(NSUInteger)theIndex {
    return [self.masterBirdSightingList objectAtIndex:theIndex];
}
- (void)addBirdSightingWithSighting:(BirdSighting *)sighting {
    [self.masterBirdSightingList addObject:sighting];
}
@end
```

Master View Controller Files

This section contains listings for the following files:

- `BirdsMasterViewController.h`

- `BirdsMasterViewController.m`

BirdsMasterViewController.h

```
#import <UIKit/UIKit.h>
@class BirdSightingDataController;
@interface BirdsMasterViewController : UITableViewController
@property (strong, nonatomic) BirdSightingDataController *dataController;
- (IBAction)done:(UIStoryboardSegue *)segue;
- (IBAction)cancel:(UIStoryboardSegue *)segue;
@end
```

BirdsMasterViewController.m

```
#import "BirdsMasterViewController.h"

#import "BirdsDetailViewController.h"
#import "BirdSightingDataController.h"
#import "BirdSighting.h"
#import "AddSightingViewController.h"

@implementation BirdsMasterViewController

- (void)awakeFromNib
{
    [super awakeFromNib];
    self.dataController = [[BirdSightingDataController alloc] init];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.rightBarButtonItem.accessibilityHint = @"Adds a new bird
sighting event";
    // Do any additional setup after loading the view, typically from a nib.
}
```



```
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

#pragma mark - Table View

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

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [self.dataController countOfList];
}

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

    static NSString *CellIdentifier = @"BirdSightingCell";

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];

    BirdSighting *sightingAtIndex = [self.dataController
objectInListAtIndex:indexPath.row];
    [[cell textLabel] setText:sightingAtIndex.name];
```



```
    [[cell detailTextLabel] setText:[formatter stringFromDate:(NSDate
*)sightingAtIndex.date]];
    return cell;
}


- (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath
*)indexPath
{
    // Return NO if you do not want the specified item to be editable.
    return NO;
}


- (IBAction)done:(UIStoryboardSegue *)segue
{
    if ([[segue identifier] isEqualToString:@"ReturnInput"]) {


        AddSightingViewController *addController = [segue sourceViewController];
        if (addController.birdSighting) {
            [self.dataController
addBirdSightingWithSighting:addController.birdSighting];
            [[self tableView] reloadData];
        }
        [self dismissViewControllerAnimated:YES completion:NULL];
    }
}
- (IBAction)cancel:(UIStoryboardSegue *)segue
{
    if ([[segue identifier] isEqualToString:@"CancelInput"]) {
        [self dismissViewControllerAnimated:YES completion:NULL];
    }
}


- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"ShowSightingDetails"]) {
        BirdsDetailViewController *detailViewController = [segue
destinationViewController];
```



```
        detailViewController.sighting = [self.dataController
objectInListAtIndex:[self.tableView indexPathForSelectedRow].row];
    }
}

@end
```

Detail View Controller Files

This section contains listings for the following files:

- `BirdsDetailViewController.h`
- `BirdsDetailViewController.m`

BirdsDetailViewController.h

```
#import <UIKit/UIKit.h>
@class BirdSighting;
@interface BirdsDetailViewController : UITableViewController

@property (strong, nonatomic) BirdSighting *sighting;
@property (weak, nonatomic) IBOutlet UILabel *birdNameLabel;
@property (weak, nonatomic) IBOutlet UILabel *locationLabel;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;
@end
```

BirdsDetailViewController.m

```
#import "BirdsDetailViewController.h"
#import "BirdSighting.h"
@interface BirdsDetailViewController ()
- (void)configureView;
@end
```



```
@implementation BirdsDetailViewController

#pragma mark - Managing the detail item

- (void)setSighting:(BirdSighting *) newSighting
{
    if (_sighting != newSighting) {
        _sighting = newSighting;

        // Update the view.
        [self configureView];
    }
}

- (void)configureView
{
    // Update the user interface for the detail item.
    BirdSighting *theSighting = self.sighting;

    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
    if (theSighting) {
        self.birdNameLabel.text = theSighting.name;
        self.locationLabel.text = theSighting.location;
       self.dateLabel.text = [formatter stringFromDate:(NSDate *)theSighting.date];
    }
}

- (void)viewDidLoad
{
    [super viewDidLoad];
```



```
    // Do any additional setup after loading the view, typically from a nib.
    [self configureView];
}
@end
```

Add Scene View Controller Files

This section contains listings for the following files:

- `AddSightingViewController.h`
- `AddSightingViewController.m`

AddSightingViewController.h

```
#import <UIKit/UIKit.h>
@class BirdSighting;
@interface AddSightingViewController : UITableViewController <UITextFieldDelegate>
@property (weak, nonatomic) IBOutlet UITextField *birdNameInput;
@property (weak, nonatomic) IBOutlet UITextField *locationInput;
@property (strong, nonatomic) BirdSighting *birdSighting;
@end
```

AddSightingViewController.m

```
#import "AddSightingViewController.h"
#import "BirdSighting.h"
@interface AddSightingViewController ()

@end

@implementation AddSightingViewController
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    if ((textField == self.birdNameInput) || (textField == self.locationInput)) {
        [textField resignFirstResponder];
```



```
    }
    return YES;
}
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Uncomment the following line to preserve selection between presentations.
    // self.clearsSelectionOnViewWillAppear = NO;

    // Uncomment the following line to display an Edit button in the navigation
bar for this view controller.
    // self.navigationItem.rightBarButtonItem = self.editButtonItem;
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[segue identifier] isEqualToString:@"ReturnInput"]) {
        if ([self.birdNameInput.text length] || [self.locationInput.text length])
 {
            BirdSighting *sighting;
            NSDate *today = [NSDate date];
            sighting = [[BirdSighting alloc] initWithName:self.birdNameInput.text
 location:self.locationInput.text date:today];
            self.birdSighting = sighting;
        }
    }
}
@end
```

Document Revision History

This table describes the changes to *Your Second iOS App: Storyboards*.

Date	Notes
2013-10-22	Moved to Retired Documents Library.
2012-10-16	Added information about Auto Layout and accessibility and changed the app to use unwind segues to return data from a destination scene.
2012-02-28	Made minor corrections.
2012-02-16	Updated for Xcode 4.3.
2012-01-09	New document that describes how to use storyboards to implement a master-detail app.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010