Fritz Anderson

Xcode 5
Start to Finish

iOS and OS X Development

Xcode 5
Start to Finish

Note

You can set the text style of multiple items by selecting them all and setting the style in the Attributes inspector.

Next, the labels for the statistics themselves. Insert five `UILabels` next to the stat-name labels. Throughout the process, Interface Builder will snap your views to blue guide lines that show standard spacing, and alignment with their neighbors. In the case of views with text content, you'll also be given a dotted line when the view's baseline is aligned with its neighbors. When in doubt, align baselines.

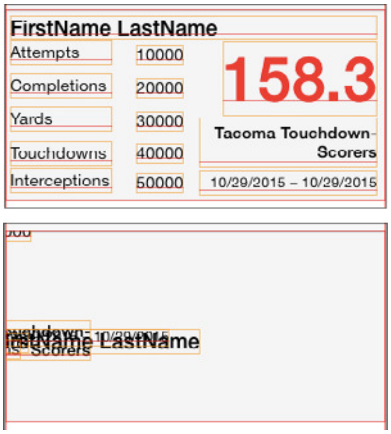


Figure 11.10 (top) The billboard view as it comes from the hard work of filling in the constituent labels. The lines around the labels are their layout rectangles, which are the criteria autolayout uses to calculate layout. (bottom) The same view, if autolayout is fired without any work on constraints.

I suggest filling the value labels with `10000`, `20000`, etc., so you can tell them apart in listings. Make their text style the same as the stat-name labels, only right-justified.

The result should look like Figure 11.10, top. I used **Editor →Canvas →Show Layout Rectangles** for clarity. Note that I've taken care to keep the labels from overlapping. Autolayout doesn't work well if views overlap their nearest neighbors.

Just for fun, make sure the Game List Controller scene is selected, and select the **Editor →Resolve Auto Layout Issues →Update All Frames in Container** command, which you can also find in the third button (|●|) of the autolayout group in the canvas. (The last part of the command name will adjust to that of the scene.)

No. Unless you're aiming at a Dadaist commentary of the predicament of the modern citizen in an era of information overload, that's not what you want (Figure 11.10, bottom). Autolayout will attempt to lay out your views, like it or not, so you have to give it some constraints so it won't wreck your display. Undo (**Edit →Undo** (⌘**Z**), did I need to tell you?) the layout.

Again making sure the game list scene is selected, select **Editor →Resolve Auto Layout Issues →Add Missing Constraints in Container**. (Xcode will adapt the command name to refer to the selected scene.) Once again, Interface Builder will take its best guesses at what you intend for the layout and create constraints accordingly. If you click around among the labels, you'll see blue i-beams and alignment strokes representing the constraints.

It's good enough for now.

The Table View

It's almost an afterthought—we don't yet have a table view to hold the individual game performances, taking the form of a `UITableView`. Typing **`table`** in the Library search field should show you a "Table View." Drag it into the lower part of the main view. (Take care not to use the "Table View Controller" instead.) You'll have to expand it to fit the available space. This isn't too hard at the sides and bottom, because Interface Builder "snaps" the edges to the bounds of the main view, but it won't give you any help with the top edge.

This is something that autolayout actually makes easier. Place the table view in the lower part of the scene, and size it so it is clear of all neighbors. Click the ├┼┤ button in the autolayout group at the bottom right of the canvas to expose the **Pin** popover. Set all four spacing fields to 0, making sure the drop-down menu for the upper spacing makes it relative to the billboard view.

Set the **Update Frames** menu in the popover to **Items of New Constraints**, and click the acceptance button at the bottom, which now has the label **Add 4 Constraints**. The table view will snap to exactly the position and size you want, with the added benefit that it will resize in sync with the rest of the scene.

Outlets

Nine of the 14 labels in the billboard view display names, dates, and statistics that are taken from the Passer Rating data store. There has to be a way to get them from the model onto the screen. In the Model-View-Controller design model, that's the job of a controller—in this case `PRGameListController`. The question then becomes, how does the view controller get to the labels?

When an Interface Builder product is loaded into a running program, an object in the program is designated as the owner—in this case it will be a `PRGameListController`. The file carries in it a list that pairs objects (your choice) to *outlets* in the owner. In modern practice, the outlets take the form of declared properties. The `@interface` flags the outlets it wants to make available with the keyword `IBOutlet`:

```
@property (weak, nonatomic) IBOutlet UILabel *datesLabel;
```

Every NIB, and every Storyboard scene, has an owner. This is an object that is external to the scene (or NIB); the loading mechanism then fills the `IBOutlet` properties with pointers to objects in the scene. Storyboards and XIBs have different treatments for owners:

- Interface Builder's editor for XIBs includes an "object" in the document outline named "File's Owner." This object does not literally exist in the XIB; Interface Builder shows it in a section of the Document Outline for placeholders. It stands in for the owner object that will load the XIB (actually its NIB product) at runtime.
- In a storyboard, each scene belongs to a `UIViewController` subclass. The controller's placeholder appears in the Document Outline, and in the bottom bar when the scene is selected, as a yellow circle with a "view" in the middle.

When you create a subclass of UIViewController, NSWindowController, or NSViewController, Xcode enables a checkbox marked **With XIB for user interface**. If you check it, Xcode will create a XIB in addition to the new class's .m and .h files. Xcode knows what the owner class will be, so it sets the class of File's Owner accordingly.

If you create a XIB alone, Xcode does not know what the class of File's Owner should be, and you have to set it yourself. Select the File's Owner icon in the Document Outline and the Identity inspector (third tab in the Inspector view in the Utility area). The first field will be a combo box to enter the name of the owner's class. The box will auto-complete as you type.

It's the same with view controllers in a storyboard: After you drag a view controller into the canvas to create a scene, the owner is identified as a plain UIViewController. You must edit the class name in the Identity inspector to point the scene at the right controller.

Back to the specifics of PRGameListController. We have to hook the controller's IBOutlets to objects in its scene. You could type in @property declarations for all the labels you'll be using. Try it: In the **Editor** control in the toolbar, click the middle segment to display the Assistant editor. This splits the Editor area in two. The left half is still Main.storyboard. Select **Automatic** in the assistant's jump bar. The Assistant view fills with the header or implementation file for PRGameListController. (If it doesn't, click the icon for the controller object, either in the outline or in the bar below the scene.) The Assistant editor can be made to track the "appropriate" counterpart of any file you are editing.

IBOutlets must be in an @interface for the controller class. You have two choices for an @interface section: The one in PRGameListController.h is the main one—it is the "official" declaration of the class. But it's not the one you want. That @interface is for a public API, and other classes have no business knowing how PRGameListController runs its views.

Instead, you want the *class extension* at the top of PRGameListController.m. It declares a supplemental API for the class, and is usually kept in the .m file so it is private to the class. The template for the detail controller included a short class extension for what is now PRGameListController. At the right end of the assistant's jump bar is a control with what ought to be a **2** between two arrowheads. Click either to cycle through the "Automatic" file set.

Somewhere before the @end of the extension, type the @property for datesLabel:

```
@interface PRGameListController ()
- (void)configureView;

@property(nonatomic, weak) IBOutlet UILabel *   datesLabel;
@end
```

Save PRGameListController.h. Now go back to the Document Outline sidebar at the left edge of the canvas (click the button in the lower-left corner if it isn't visible), and right-click on "Game List Controller," the Englished name of the scene's owner. A small black heads-up display (HUD) window appears, containing a table of outlets, among them datesLabel. There's a bubble at the right end of that row in the table. Drag from it to the label you set up with a range of dates, and release; the adjoining column in the HUD fills with a reference to the label (Figure 11.11). Now, when the scene is loaded, the owning PRGameListController's datesLabel property will contain a pointer to that label.

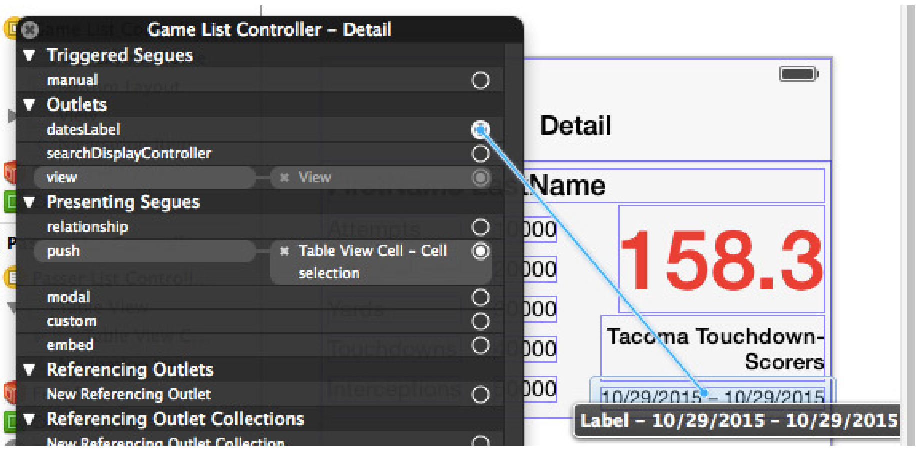
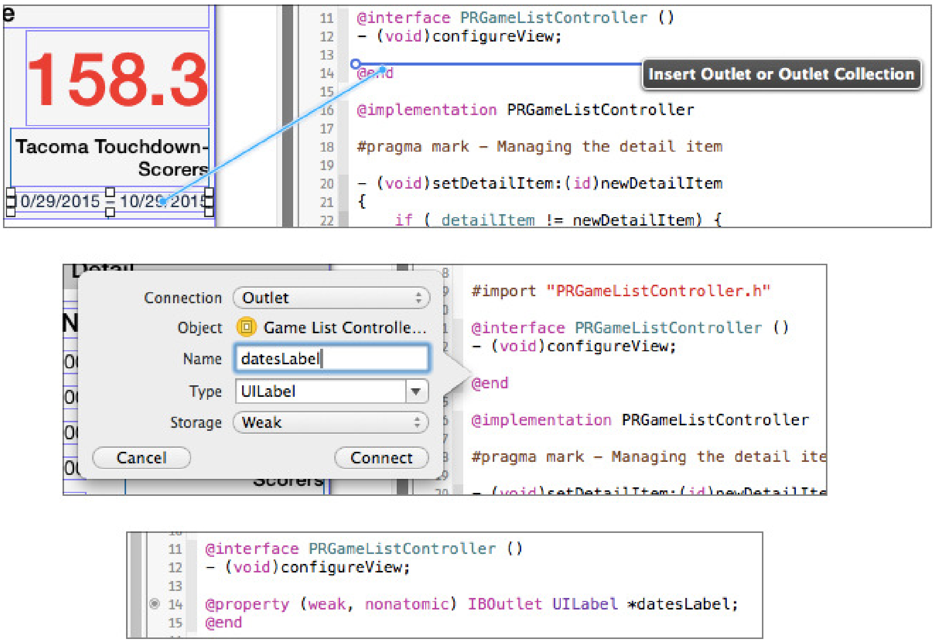


Figure 11.11 Right-clicking the "Game List Controller" entry in the Document Outline opens a heads-up display window that includes the controller's outlets. Drag from an outlet's linkage bubble to a view in the controller's scene, and the outlet's @property will be filled with a pointer to that view.

You could repeat this with each IBOutlet you will need for the data labels: Declare the property, make sure it has the right type, open the HUD, and drag out the connections. Not too bad, but there is a better way.

In the HUD, click the **x** next to the outlet to clear the link. In the header, delete the datesLabel @property. Now control-drag from the date-range label into the class-extension @interface. As you drag within the @interface, a horizontal insertion bar appears. Let go of the drag. Xcode displays a popover window. This window offers to declare a @property where you let go of the drag. All you have to do is type **datesLabel** into the **Name** field, click **Connect**, and the declaration appears. See Figure 11.12.



```
@interface PRGameListController ()
- (void)configureView;

@end

@implementation PRGameListController

#pragma mark - Managing the detail item

- (void)setDetailItem:(id)newDetailItem
{
    if ( detailItem != newDetailItem) {
```

Connection: Outlet
Object: Game List Controlle...
Name: datesLabel
Type: UILabel
Storage: Weak

```
#import "PRGameListController.h"

@interface PRGameListController ()
- (void)configureView;

@end

@implementation PRGameListController

#pragma mark - Managing the detail ite
```

```
@interface PRGameListController ()
- (void)configureView;

@property (weak, nonatomic) IBOutlet UILabel *datesLabel;
@end
```

Figure 11.12 (top) Control-drag from a view in a scene into its owner's `@interface` in the Assistant editor. (middle) Releasing the drag produces a popover offering to declare an `IBOutlet` to the view. (bottom) Click **Connect** to declare the outlet and link it to the view.

As we saw with the short-lived `detailDescriptionLabel` outlet in the "Building a View" section earlier in this chapter, the connection bubble for the new outlet is filled both in the source code and in the Connections inspector, and if you hover the mouse over them, the dates label will highlight in the storyboard.

Hooking Up the Outlets

Before you go on a spree of making outlet connections, taking a little care will pay off: There is a + button next to the right end of the assistant's jump bar. Click it. The Assistant area is now divided into two editors. Use the lower editor's jump bar to navigate to `_PRPasser.h`. You want the names of the new outlets to match up with the names of the `_PRPasser` properties they display, and the new editor will give you a reference for the property names.

Now control-drag from the variable labels to make new properties in `PRGameListController`. Use this convention in naming the outlets: Take the name of a Passer property, and add **`Label`** to it. The team name label goes into the controller interface as **`currentTeamLabel`**, attempts as **`attemptsLabel`**, and so on.

Note

Interface Builder can also link controls to action methods, declared with the `IBAction` tag in the class `@interface`. When you trigger a control that you've linked to an `IBAction`, the action method is executed. See the "Wiring a Menu" section of Chapter 18, "Starting an OS X Application," for an example.

`PRGameListController` needs to know about the table view, as well. Control-drag a link from the table to the `@interface`, and name it `tableView`.

Checking Connections

Do one last pass to verify that everything is connected to what it's supposed to connect to: With the storyboard in the main editor, and the `IBOutlet` declarations for `PRGameListController` showing in an Assistant editor, run your mouse down the connection dots, and make sure every view gets highlighted in turn.

If an outlet isn't connected, or is connected to the wrong view, drag from the connection bubble to the correct view. An `IBOutlet` can refer to at most one view—it's just a single pointer. A view can be connected to many outlets because it has no reference back to the outlets. Checking the outlets, one by one, and reconnecting them as needed will be enough to get you out of any tangles.

Connecting `PRGameListController`

Interface Builder is great, but you still have to write code to get data from the model into the view. You'll make some changes to `PRGameListController`.

The template provides a setter for `detailItem`, `setDetailItem:`, that calls through to a `configureView` method. That's where you'll move the statistics in Passer to the labels in the view.

The `configureView` Method

Here's `configureView`. It's a little long, but there's a point I want to make:

```
- (void)configureView
{
    if (self.detailItem) {
        PRPasser *      thePasser = self.detailItem;
        NSArray *   integerProperties = @[
                          @"attempts",
                          @"completions",
                          @"yards",
                          @"touchdowns",
                          @"interceptions"];
        for (NSString * prop in integerProperties) {
            //  Get the property from the PRPasser
            //  valueForKey: will wrap it as an NSNumber object
```