

2 iPhone app patterns

Hello @twitter!



Apps have a lot of moving parts. OK, actually, they don't have any real moving parts, but they do have lots of UI controls. A typical iPhone app has more going on than just a button, and it's time to build one. Working with some of the more complicated widgets means you'll need to pay more attention than ever to how you design your app, as well. In this chapter, you'll learn about some of the fundamental design patterns used in the iPhone SDK, and how to put together a bigger application.



Author's note:

Head First does not take any responsibility for Mike's relationship problems.

Mike is back. He has a great girlfriend, Renee, but they've been having some problems. She thinks that he doesn't talk about his feelings enough.



Mike

A Twitter app is the way to go here.
That would be perfect: I can just tweet about my feelings and then she'll be happy.

There's (about to be) an app for that.
Using some solid design and the basic controls included in the Interface Builder library, you can have Mike posting to Twitter in no time. But first, what should his tweets say?

First we need to figure out what Mike (really) wants

Mike isn't a complex guy. He wants an easy interface to talk to Twitter and he really doesn't want to have to type much.

Here's what Mike handed you at the end of the night

Here's what I want:

- Not much typing
- Instant communication
- Easy to use
- My tweets like this: I'm _____ and feeling _____ about it."



App Magnets

Now that we know what Mike wants, what do we need to do? Take the magnets below and put them in order of the steps you'll follow to build his Twitter app.

Determine app layout

Build the GUI

Figure out how to use the controls

Handle the data

Send output to Twitter



App Magnets Solution

Now that we know what Mike wants, what do we need to do? Take the magnets below and put them in order of the steps you'll follow to build his Twitter app.

After you've landed on the general app design, you need to get into the documentation a little and figure out how to implement the controls you've chosen.

Here we need to manage the data coming from the controls...

Determine app layout

Build the GUI

Figure out how to use the controls

Handle the data

Send output to Twitter

Before you start coding anything, sketch up what you're thinking.

Some people write backend code first – we're going to go back and forth depending on our project, but to get started, we'll do the GUI first this time.

We'll help you out with this last step, too.

there are no Dumb Questions

Q: How do you figure out the app layout?

A: We're going to give you a couple to choose from to get started, but in general, it's important to think about what your app needs to do and focus on those features first.

Q: Are we always going to start with a sketch?

A: Yes! Good software design starts with knowing what you're building and how

the user is going to work with the app. The app for Mike is going to work with Twitter, and he's going to be able to make some selections for his feelings and thoughts. That's it!

Q: How do we talk to Twitter?

A: Don't worry, we'll give you some code to help you to work with that.

Just FYI, though, Twitter has a really well-documented API. We'll give you what you need, but feel free to add more features!

Q: Does every control work differently than the others?

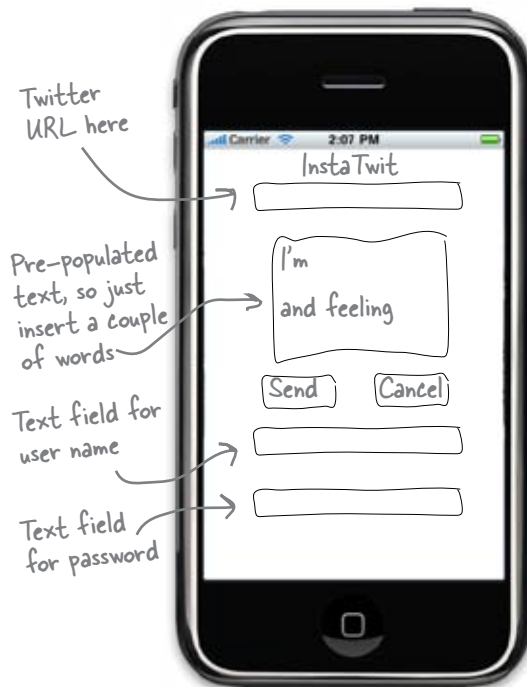
A: For the most part, no—once you learn a few basic patterns, you'll be able to find your way through most of the SDK. Some of the controls have a few peculiarities here and there, but for the most part they should start to look familiar.

APP LAYOUT CONSTRUCTION

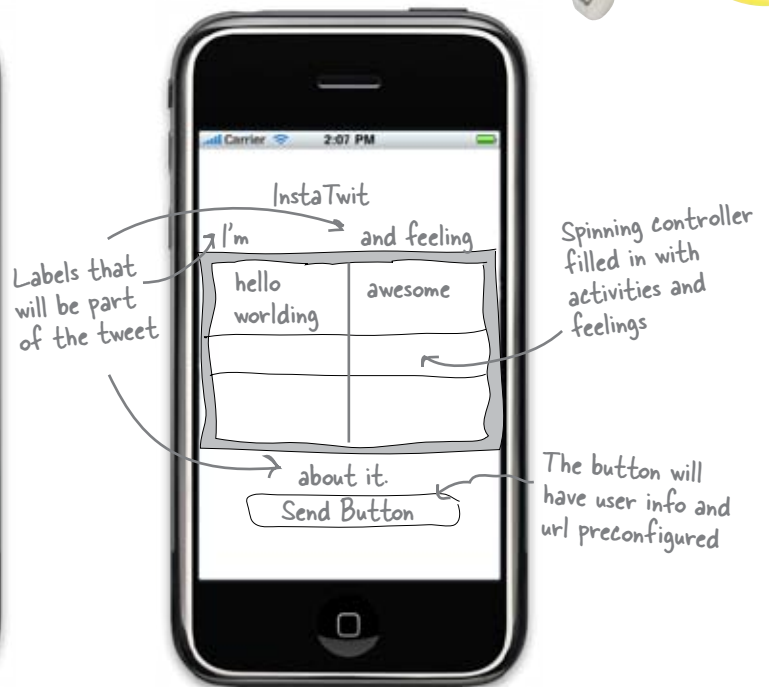
Here are two designs to evaluate. Based on aesthetics, usability, and standard iPhone app behavior, which one is better for Mike?



Option #1



Option #2



Which app is better?

Why? (Be specific.)

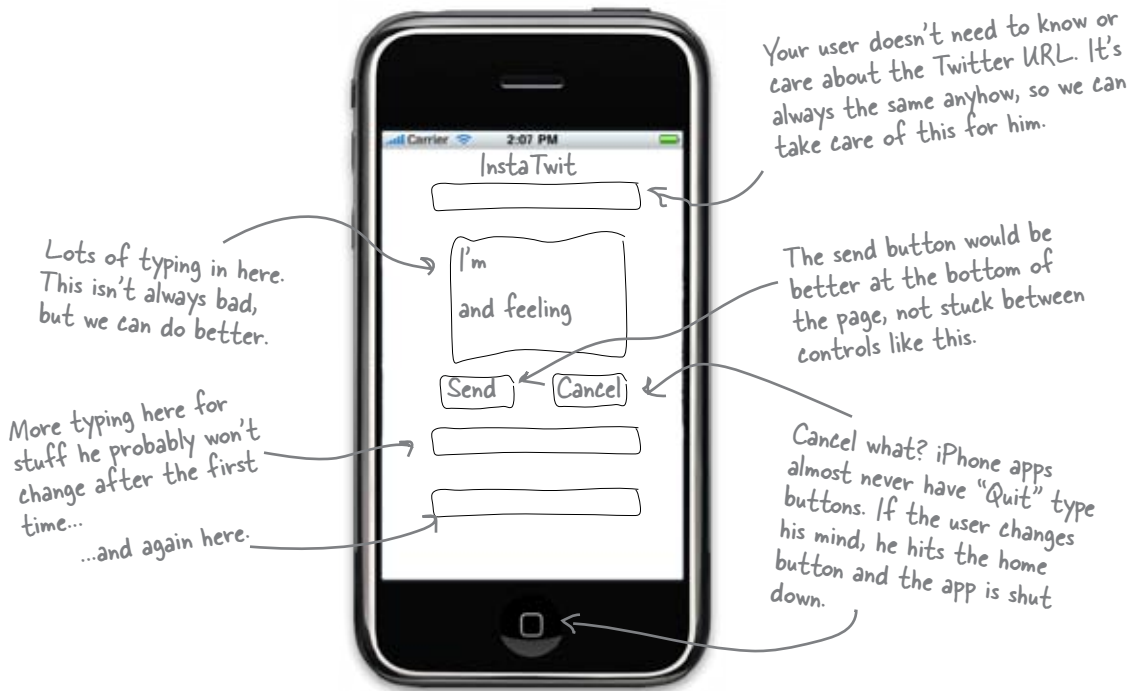
.....

Why not the other?

APP LAYOUT CONSTRUCTION

We've given you two designs to evaluate. Based on aesthetics, usability, and standard iPhone app behavior, which one is better for Mike?

Bad Option #1

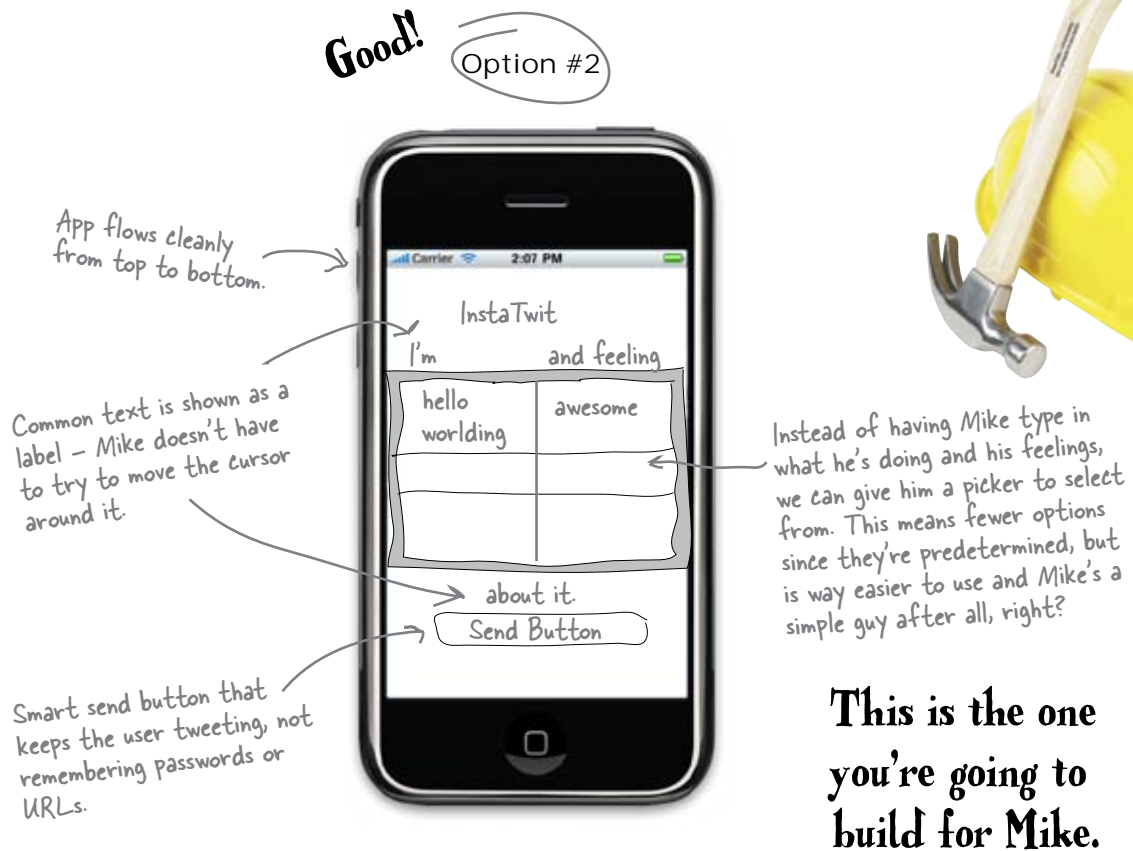


Which app is better? #2.

Why? (Be specific.) Option #2 has a lot less typing and fewer fields overall.

Since the user doesn't need to change his username or password often there's no reason to put it on the main view every time he runs the app.

Why not the other? Option #1 has a lot of typing and settings to remember. The buttons are confusing.



there are no Dumb Questions

Q: Do I really need to care about usability and aesthetics so much?

A: Usability and aesthetics are what made the iPhone a success, and Apple will defend them to the death. Even more importantly, you don't get to put anything on the App Store or on anyone else's iPhone without their approval. Apple has sold over a billion apps—if yours doesn't fit with the iPhone look and feel or is hard to use, people will find someone else's app and never look back.

Q: We got rid of the username, password, and URL fields. The URL one I understand, but what about the other two?

A: Anytime your app needs configuration information that the user doesn't need to change frequently, you should keep it out of the main task flow. Apple even provides a special place for these called a Settings bundle that fits in with the standard iPhone settings. We're not going to use that in this chapter (we'll just hardcode the values) but later we'll show you how to put stuff in the Settings page. That's usually the right place for things like login details.

Q: How am I supposed to know what Apple thinks is good design or aesthetically pleasing?

A: Funny you should ask... go ahead, turn the page.

App design rules—the iPhone HIG

The iPhone Human Interface Guide (HIG) is a document that Apple distributes for guidance in developing iPhone Apps for sale on the App Store. You can download it at <http://developer.apple.com/iphone>. This isn't just something nice they did to help you out; when you submit an app for approval, you agree that your app will conform to the HIG.

We can't overstate this: **you have to follow the HIG**, as Apple's review process is thorough and they will reject your application if it doesn't conform. Complain, blog with righteous anger, then conform. Now let's move on.

Apple also distributes a few other guides and tutorials, including the iPhone Application Programming Guide. This is another great source of information and explains how you should handle different devices, like the iPhone and the iPod Touch. Not paying attention to the iPod Touch is another great way to get your app rejected from the App Store.

Note: While the authors do not suggest testing these methods of being rejected from the App Store, we can speak with authority that they work.

Application types

The HIG details three main types of applications that are commonly developed for the iPhone. Each type has a different purpose and therefore offers a different kind of user experience. Figuring out what type of application you're building before you start working on the GUI helps get you started on the road to good interface design.

Immersive Apps



Games are a classic example, but like this simulated level, all immersive apps require a very custom interface that allows the user to interact with the device. As a result, HIG guidelines aren't as crucial in this case.

Productivity Apps

Help manage information and complete tasks. Info is hierarchical, and you navigate by drilling down into more levels of detail.



Utility Apps

Get a specific set of info to the user with as little interaction or settings configuration as possible.



Usually have more interface design than a productivity app, and are expected to stay very consistent with the HIG.

★ WHO DOES WHAT? ★

Below are a bunch of different application ideas. For each one, think about what kind of app it really is and match it to the app types on the right.

App Description

InstaTwit 1.0: Allows you to tweet with minimal typing.

News Reader: Gives you a list of the news categories and you can get the details on stories you choose.

Marble Game: A marble rolling game that uses the accelerometer to drive the controls.

Stopwatch Tool: Gives you a stopwatch that starts and stops by touching the screen

Recipe Manager: A meal listing that allows you to drill down and look at individual recipes.

Type of App

Immersive Application

Utility Application

Productivity Application

WHO DOES WHAT? SOLUTION

Match each app description to its application type.

App Description

Type of App

InstaTwit 1.0: Allows you to tweet with minimal typing.

Since we have one screen and no typing, InstaTwit is more of a Utility App

News Reader: Gives you a list of the news categories and you can get the details on stories you choose.

Since this App has a list-driven, drill-down interface, it's Productivity

Marble Game: A marble rolling game that uses the accelerometer to drive the controls.

Using the accelerometer as the control and a big rolling marble...

Stopwatch Tool: Gives you a stopwatch that starts and stops by touching the screen

We want a very focused stopwatch GUI, no real data to work through

Recipe Manager: A meal listing that allows you to drill down and look at individual recipes.

Lots of data to work through here: tables, a drill-down to recipes—definitely productivity

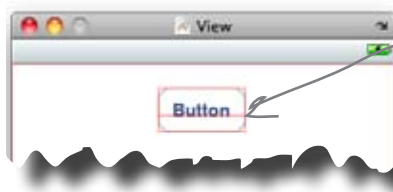
Immersive Application

Utility Application

Productivity Application

HIG guidelines for pickers and buttons

The HIG has a section on the proper use of all the standard controls, including the two that we've selected for InstaTwit. Before you build the view with your controls, it's a good idea to take a quick look at the recommendations from Apple. You'll find this information in Chapter 9, Application Controls, of the HIG.



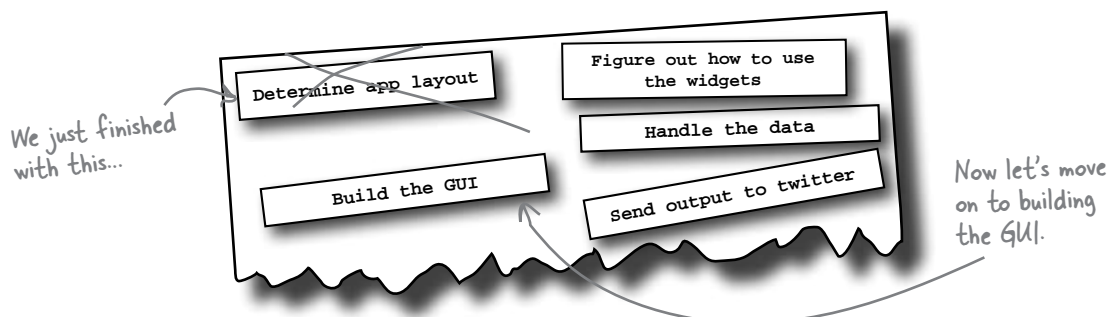
The rounded rectangle button is pretty straightforward, but keep in mind it should always perform some kind of action.



The picker only displays a few items on the screen at a time, so remember that your user isn't going to be able to see all the options at once.

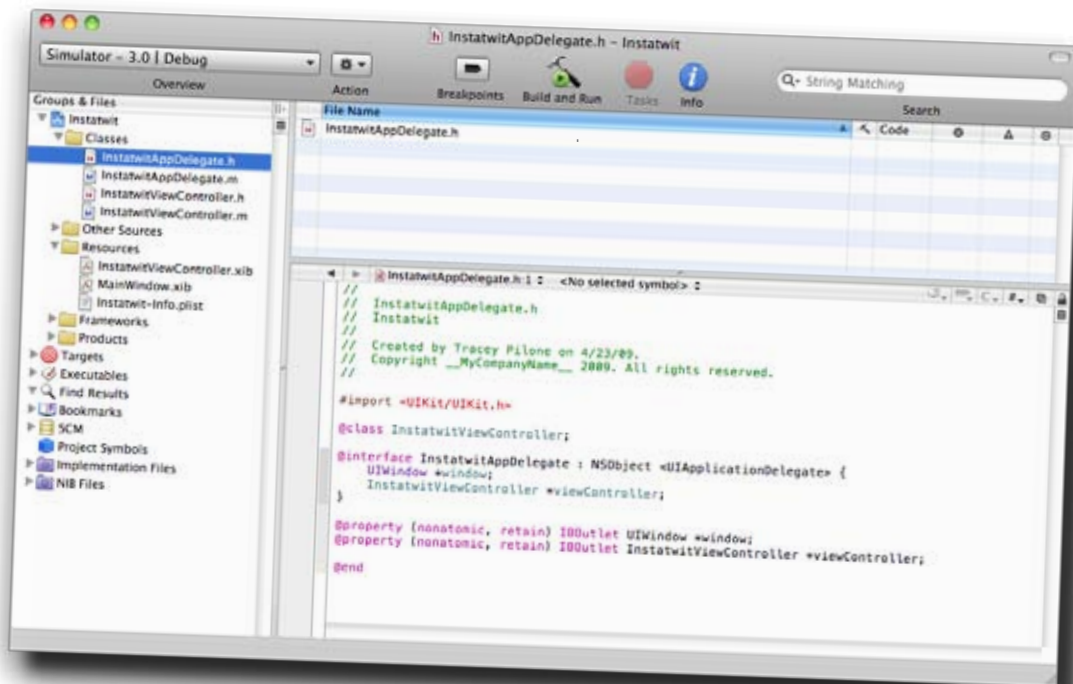
If you have units to display, they need to be fixed to the selection bar here.

The picker's overall size is fixed, although you can hide it or have it be part of the view (like we do in InstaTwit).



Create a new View-based project for InstaTwit

Once you've started Xcode, select **File** → **New Project**. Just like iDecide, for InstaTwit we have one screen and we're not going to be flipping it or anything fancy, so again choose the **View-based Application** and name it InstaTwit.



Watch it!

The new project type is not necessarily the same as your app type.

For example, a Productivity App can be written as a View-based Application, a Window-based Application, Navigation-based Application, or a Tab Bar Application.

We'll be working with these other project types later in the book.

Start with the view layout

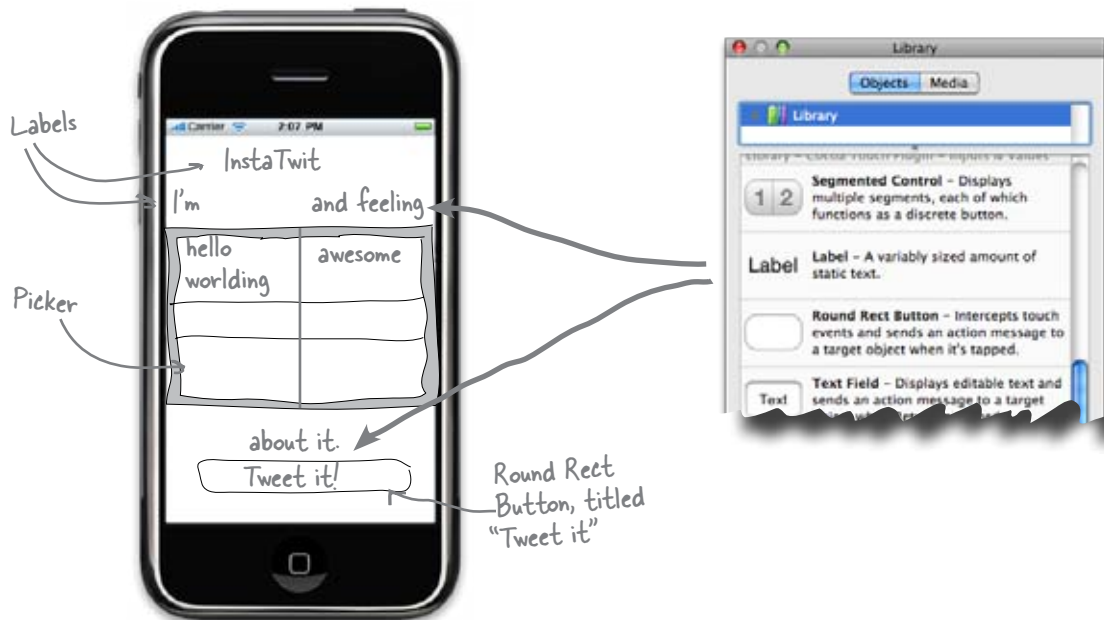
Now that we have the autogenerated code, we're going to start working with the interface. To do that, we'll be editing the nib (.xib) file. Double-click on InstaTwitViewController.xib in the Resources folder, and launch Interface Builder.



Exercise

It's time to build the View. Using drag and drop, pull over the elements from the Interface Builder library that you need to build the view.

- 1 Find each of the elements (we've given them the proper name for you) in the library and drag and drop them into the **View** window.



- 2 Select the top label and hit ⌘1. That will launch the **Inspector**.

Edit label text here



- 3 Edit the labels and button text for the title, "I'm", "and feeling", and "about it", as well as the title for the button. Don't worry about the picker values just yet.

Once you save it, your view should look like this...



Exercise Solution

The View is all built and ready to go. Here's what you should have on your screen now. Once you tweak everything to look just how you want it, we'll run InstaTwit.



Your labels may not be this big. By default, the label will not resize to the font, but to fit the space. To make it larger, just resize using the dots at the edges of the label field.

Filling in the picker data requires some code, and we'll get to that in a minute. What you see here are default values.



The inspector for the button is slightly different—the title is further down in the window.

Did you notice the blue guidelines in the simulator? They're in the view when you're laying out elements to help you center things and keep them lined up with each other.



Test Drive

Now it's time to check out InstaTwit in the Simulator. Save in Interface Builder, go back into Xcode, and hit **Build and Debug** from the Build menu (or ⌘ return).

The picker isn't showing up because there isn't any data yet...



To get the picker to show, it needs to have data to fill it. Where do you think that the code for the data should go?

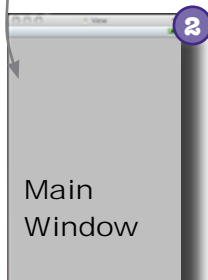
The life of a root view

In Chapter 1 we touched on how Interface Builder creates XML descriptions of your view, called a nib, and that the Cocoa Touch framework turns that into a real view in your application. Now that you've built a couple apps, let's take a closer look at what's going on under the hood.

1

Like in most other languages, `main(...)` gets called first.

When your application is launched by the user, the iPhone provides a quick animation of your app zooming into the screen (this is actually a PNG file you can include with your app), then calls your main method. Main is provided by the templates and you almost never need to touch it.



2

Main kicks off a Cocoa Touch Application.

The standard `main(...)` kicks off a Cocoa Touch `UIApplicationMain`, which uses the information in your application's `Info.plist` file to figure out what nib to load. With the View template we used, it's a nib called `MainWindow.xib`.

3

`MainWindow.xib` contains the connections for our application.

If you look in `MainWindow.xib`, you'll see it has an instance of our `InstaTwitAppDelegate`, for its `UIApplicationDelegate` and an instance of our `InstaTwitViewController`. When the Cocoa framework loads this nib, it will create an instance of our `InstaTwitViewController` and tell it to load our `InstaTwitViewController.xib`.



This is the View Controller. It subclasses `UIViewController`.

`InstaTwitViewController` instantiated from `MainWindow.xib`

When we built the nib, we used the generic proxy File's Owner for outlet and action connections. When the nib is actually loaded, there's a real object there to receive those connections. For us, it's the `InstaTwitViewController`.

We'll talk more about delegates soon, too.

4

The Cocoa Touch framework creates our custom view from the `InstaTwitViewController.xib`.

When we constructed the nib, we used the File's Owner proxy object to stand in for the object that owns the nib contents. At this point the framework is loading the nib on behalf of our `InstaTwitViewController` class so that instance is used for connections. As the framework creates instances of our components, they're connected up to the instance of `InstaTwitViewController`.

This is our view.



The nib file contains serialized instances of objects as we configured them. They are usually control objects like buttons or labels, but can be anything that can be serialized.

5

When events occur with components, methods are invoked on our controller instance.

The actions we associated between the controls and the File's Owner in the nib were translated into connections between the controls and our instance. Now when a control fires off an event, the framework calls a method on our `InstaTwitViewController` instance.

Now let's put this knowledge to use and add some data for the picker.

there are no Dumb Questions

Q: Isn't good design vs. bad design a little subjective?

A: Yes and no. Obviously, different people will have differing opinions about what UI looks better. However, Apple has very specific guidelines about how certain controls should be used and best practices that should be followed. In general, if you're using a common iPhone control, make sure you're using it in a way that's consistent with existing applications.

Q: How can I run these apps on my iPhone?

A: To get an app you write installed on your iPhone you'll need to sign up for either the Standard or Enterprise Developer programs at <http://developer.apple.com/iphone/>. Everything in this book is designed to work with just the **Simulator**, so don't feel like you need to go do that just yet. We'll talk more about putting apps on an actual phone later in the book.

Q: The InstaTwit icon looks horrible. What can I do?

A: The icon for an application is just a PNG file in your project. We'll add and configure icons later, but for now, just know that you'll need a .png file in the resources directory for that purpose—we'll hook you up with some cool icons later.

Q: Do I have to use Interface Builder for the view?

A: No. Everything that you do in Interface Builder can be done in code. Interface Builder makes it a lot easier to get things started, but sometimes you'll need that code-level control of a view to do what you want. We'll be switching back and forth depending on the project and view.

Q: I'm still a little fuzzy on this nib thing. Do they hold our UI or regular objects?

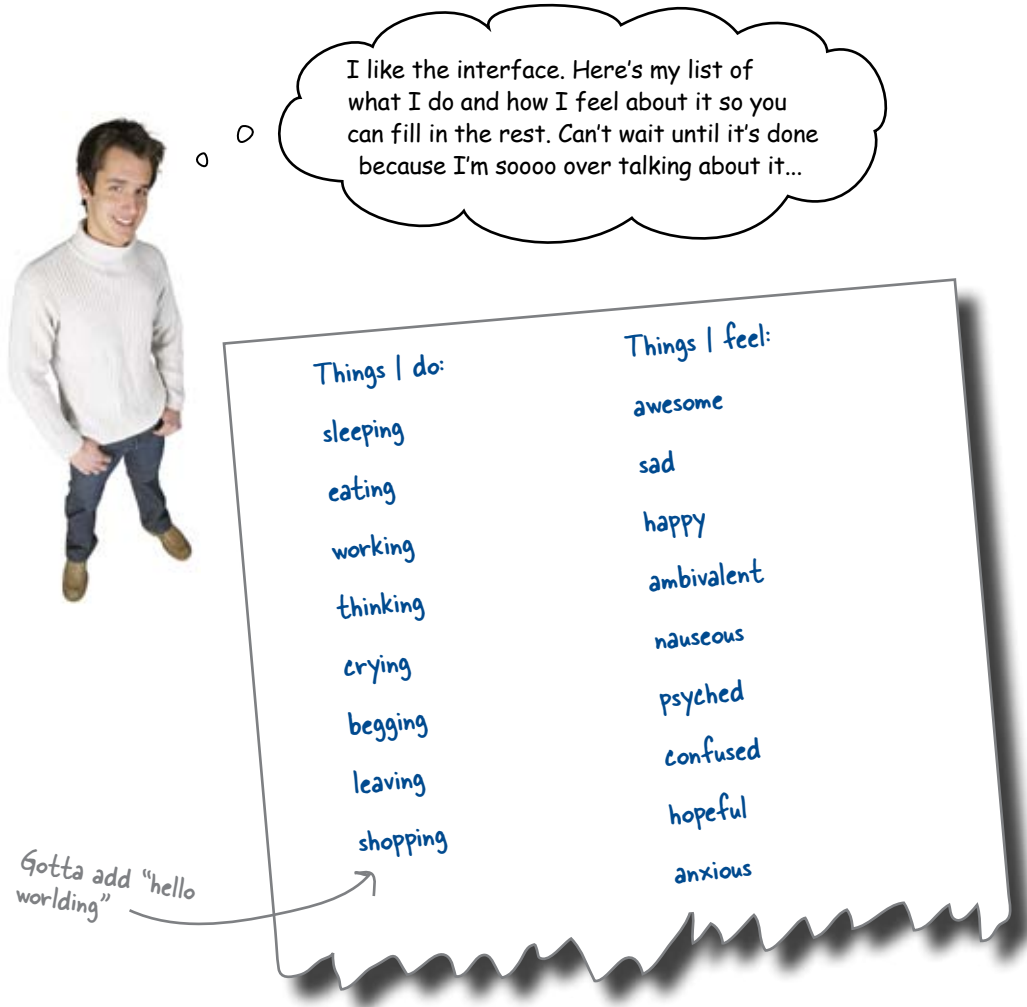
A: They can hold both. When you assemble a view using Interface Builder, it keeps track of the controls you're using and the links to other classes. These controls are serialized into an XML document; when you save it out, this is your nib. Interface Builder is able to serialize non-control classes, too. That's how it saves out our `InstaTwitViewController` in `MainWindow.xib`. When the nib is restored from disk, objects in the nib are instantiated and populated with the values you gave them in Interface Builder.

Q: So does Interface Builder save out the File's Owner too?

A: No, File's Owner is a proxy. File's Owner represents whatever class is asking to have this nib loaded. So the File's Owner proxy isn't actually stored in the nib, but Interface Builder needs that proxy so you can make association with controls you used in your view. When the nib is restored (and the control objects are instantiated), the nib loading code will make the connections to the real owning object that asked to load the nib.

First, get the data from Mike

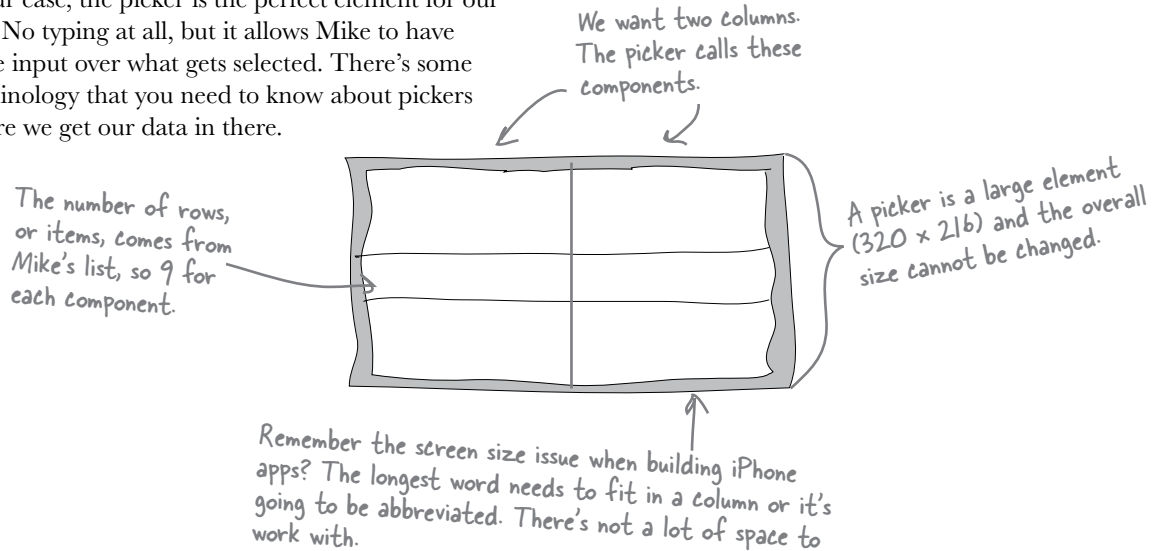
Mike likes what you have put together for the UI, so now we need a little more information before we fill the picker.



This data will be used as part of the picker, but how do you implement that?

Use pickers when you want controlled input

In our case, the picker is the perfect element for our app. No typing at all, but it allows Mike to have some input over what gets selected. There's some terminology that you need to know about pickers before we get our data in there.

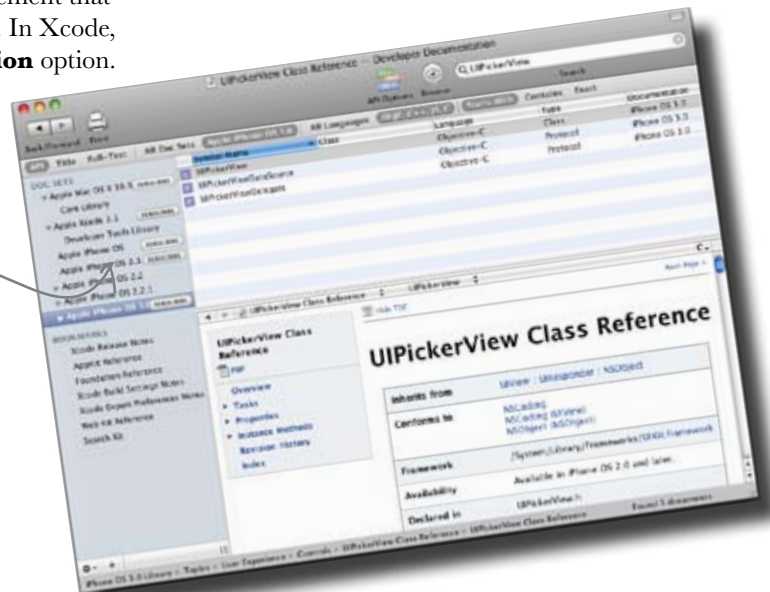


When in doubt, check out Apple's API documentation

By now you're already thinking about how to implement that picker. It's time to get into the API documentation. In Xcode, go to the **Help** menu and then the **Documentation** option.


You'll need to subscribe to the Apple iPhone OS 3.X Doc Set to keep up to date.

Search for "UIPickerView" and it will pull up all the information on the class that you need to implement for the picker.



Fill the picker rows with Mike's data

The picker needs to know how many rows it needs and how many columns. And that information is tied to the words that Mike provided.



OK, so we can just set the picker rows with the values Mike gave us like we did with the button label, right?

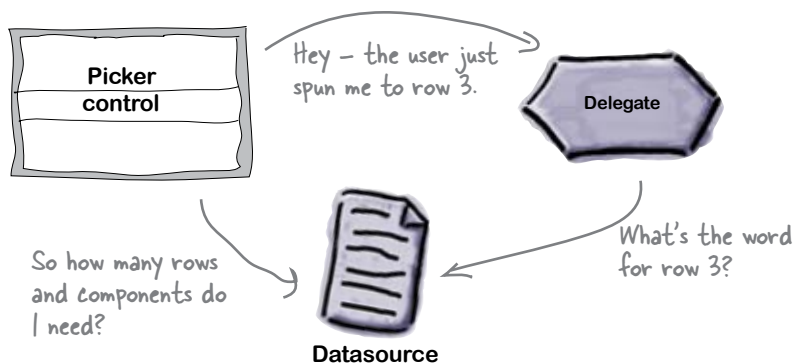
The picker is different.

The picker doesn't want to be told what to do, it's going to **ask** when it wants your input. You're going to see this pattern show up with controls that could use a lot of data like pickers and later, table views. Let's take a closer look...

Pickers get their data from a datasource...

Most of the elements in the Cocoa Touch framework have the concept of datasources and delegates. Each UI control is responsible for how things look on the screen (the cool spinning dial look, the animation when the user spins a wheel, etc.), but it doesn't know anything about the data it needs to show or what to do when something is selected.

The **datasource** provides the bridge between the control and the data it needs to display. The control will ask the datasource for what it needs and the datasource is responsible for providing the information in a format the control expects. In our case, the datasource provides the number of components (or columns) for the picker and the total number of rows for the picker. Different controls need different kinds of datasources. For the picker, we need a **UIPickerViewDataSource**.



...and tell their delegates when something happens.

A **delegate** is responsible for the behavior of an element. When someone selects something—or in this case, scrolls the picker to a value—the control tells the delegate what happened and the delegate figures out what to do in response. Just like with datasources, different controls need different kinds of delegates. For the picker, we need a **UIPickerViewDelegate**.

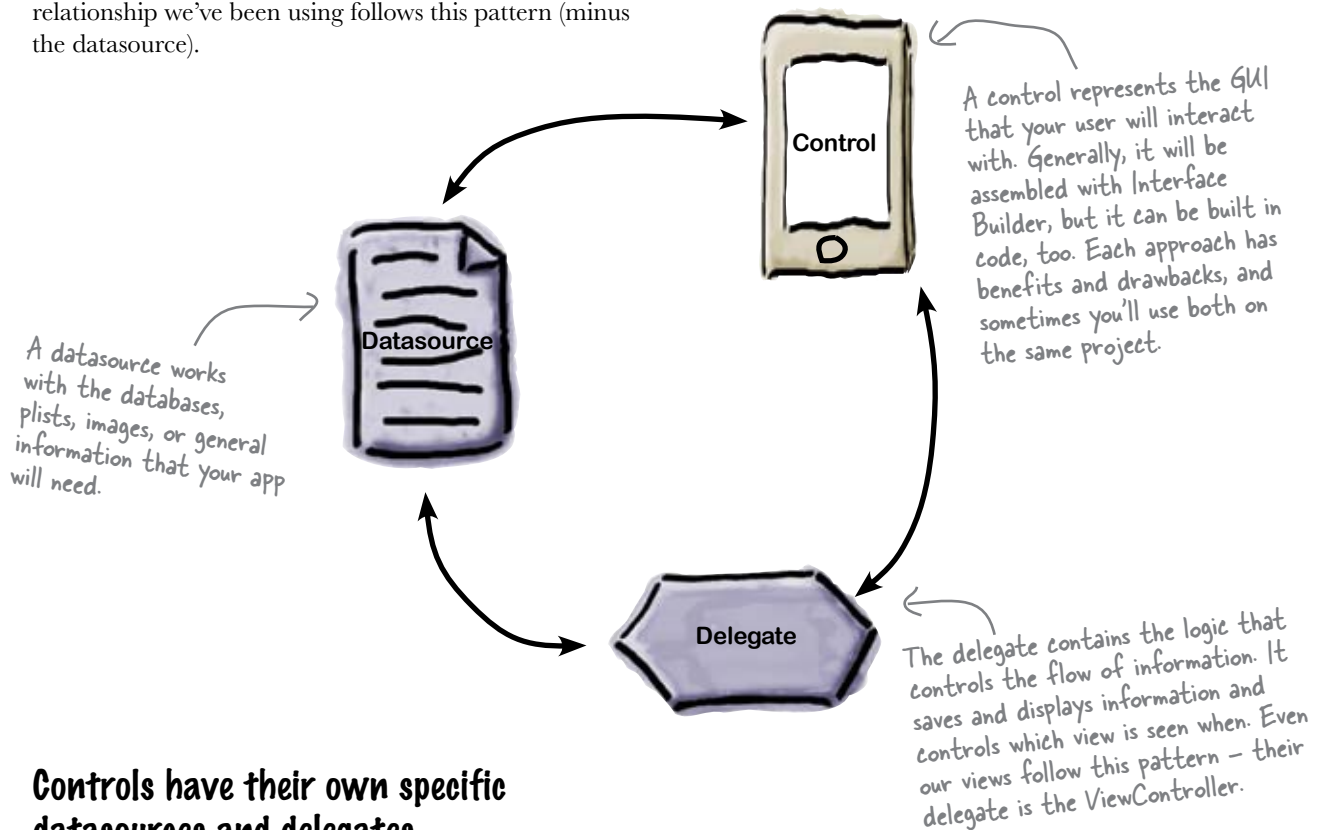
there are no
Dumb Questions

Q: Why is the delegate providing the content? That really seems like data.

A: That's something particular to a picker and it has to do with the fact that the picker delegate can change how the data is shown. In the simplest form, it can just return strings to the picker. If it wants to get fancy, it can return the entire view (yes, just like the view you built with Interface Builder, but smaller) to use images or special fonts, whatever.

There's a pattern for that

You're going to see this **Control-Datasource-Delegate** pattern show up throughout the rest of this book. Nearly all of the complex controls use it. If you squint a little, even the **View-View Controller** relationship we've been using follows this pattern (minus the datasource).



Controls have their own specific datasources and delegates

Each control has specific needs for its datasource and delegate and we'll talk about how that's handled in Objective-C in a minute. However, it's important to realize that while the responsibilities are split between the datasource and the delegate in the pattern, *they don't necessarily have to be implemented in different classes*. The control wants a delegate and a datasource—it doesn't care whether they're provided by the same object or not: it's going to ask the datasource for datasource-related things and the delegate for delegate-related things.

Let's take a closer look at how the UIPickerView uses its datasource and delegate to get an idea of how all of this fits together.



The Picker Exposed

This week's interview:
How to avoid spinning out of control...

Head First: Hello Picker, thanks for joining us.

Picker: My pleasure. I don't usually get to talk to anyone but my datasource and delegate so this is a real treat.

Head First: I'm glad you brought those up. So we've worked with controls like buttons and labels, but they just have properties. What's going on with this delegate and datasource business?

Picker: Well, to be clear, I have properties too—there just isn't too much exciting going on there. What makes me different is that I could be working with a lot of data. I might only have one row or I might have a hundred; it just depends on the application.

Head First: Ah, OK. A label only has one string in it, so there can be a property that holds that string. No problem.

Picker: Exactly! So, instead of trying to cram all of the data into me directly, it's cleaner to just let me ask for what I need when I need it.

Head First: But you need to ask for it in a specific way, right?

Picker: That's the beauty of my setup. I ask for what I need to know in a specific way—that's why there's a `UIPickerDatasource`—but I don't care where my datasource gets its information. For example, I need to know how many rows I need to show, so I ask my datasource. It could be using an array, a database, a plist, whatever—I don't care. All I need to know is how many rows.

Head First: That's really nice—so you could be showing data coming from just about anything, and

as long as your datasource knows how to answer your questions, you don't care how it stores the data internally.

Picker: You got it. Now the delegate is a little different. I can draw the wheels and all that, but I don't know what each application wants to do when someone selects a row, so I just pass the buck to my delegate.

Head First: So whichever one implements the delegate, it codes things so that when you tell it what happened, it performs the right action, like saving some value or setting a clock or whatever...

Picker: That's it. Now, I have to confess I have one little oddity going on...

Head First: Oh, I was waiting for this... this is where you ask the delegate for the value to show in a row, right?

Picker: Yeah—other controls ask their datasource. I could come up with a lot of excuses, but... well, we all have our little quirks, right?

Head First: I appreciate your honesty. It's not all bad, though; your delegate can do some neat things with each row, can't it?

Picker: Oh yeah! When I ask the delegate for a particular row, it can give me back a full view instead of just a string. Sometimes they have icons in them or pictures—really, anything you can cram in a view, I can display.

Head First: That's great. Well, we're out of time, but thanks again for stopping by.

Picker: My pleasure! Now I'm off to take my new datasource for a spin...

WHO DOES WHAT?

Match each picker characteristic to where it belongs—the delegate or the datasource. You'll need to go digging in the API to figure out where the three methods go.

Picker characteristic (or method)

Delegate or datasource?

Directions for drawing the view
for the items

The number of components

Delegate

`pickerView:numberOfRowsInComponent`

`pickerView:titleForRow:forComponent`

Datasource

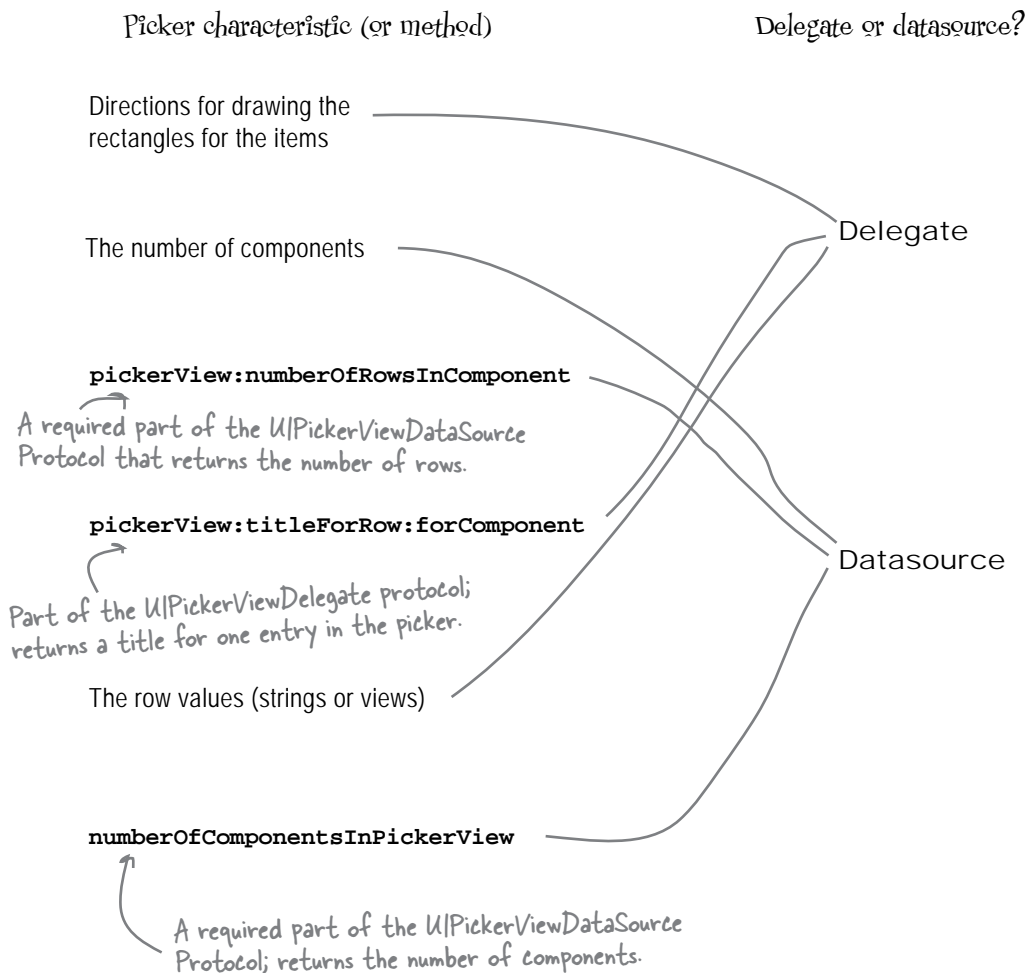
The row values (strings or views)

`numberOfComponentsInPickerView`

**Working together, the delegate
and the datasource provide what
is needed to render the picker.**

★ WHO DOES WHAT? ★ SOLUTION

Match each picker characteristic to where it belongs—the delegate or the data source. You'll need to go digging in the API to figure out where the three methods go.

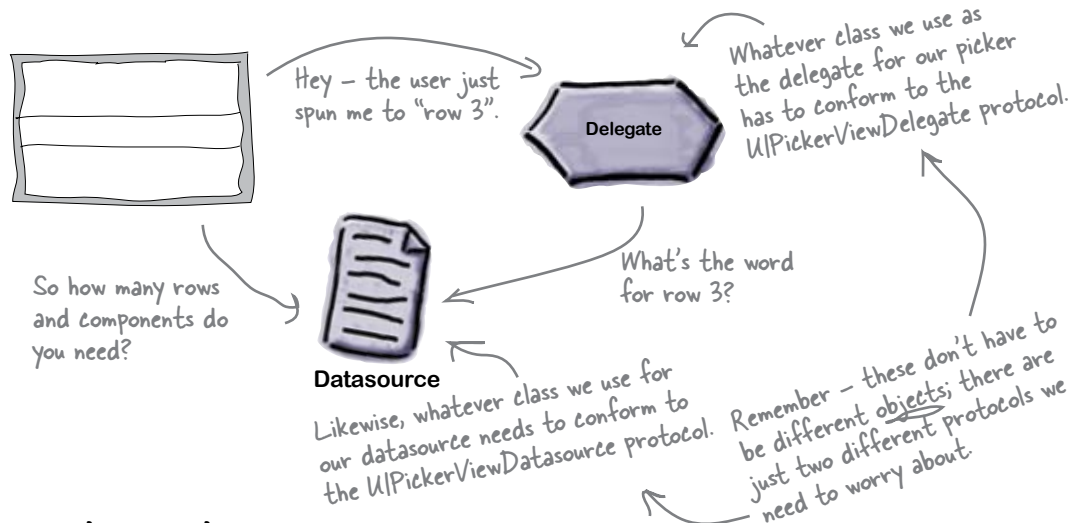


Hang on—there are protocols in both the datasource and the delegate?



Protocols define what messages the datasource and delegates need respond to.

Pickers (and other controls that use delegates and datasources) have specific messages to which their supporting classes need to respond. These messages are defined in *protocols*. Protocols are Objective-C's idea of a pure interface. When your class can speak a particular protocol, you're said to **conform to it**.



Protocols tell you what methods (messages) you need to implement

Protocols typically have some required methods to implement and others that are optional. For example, the `UIPickerViewDatasource` protocol has a required method named `pickerView:numberOfRowsInComponent`; it has to be in the datasource for the picker to work. However, `UIPickerViewDelegate` protocol has an optional method named `pickerView:titleForRow:forComponent`, so it doesn't need to be in the delegate unless you want it.

So how do you know what protocols you need to worry about? The documentation for an element will tell you what protocols it needs to talk to. For example, our `UIPickerView` needs a datasource that speaks the `UIPickerDataSource` protocol and a delegate that speaks the `UIPickerDelegate` protocol. Click on the protocol name and you'll see the documentation for which messages are optional and which are required for a protocol. We'll talk more about how to implement these in the next chapter; for now, we'll provide you the code to get started.

First, declare that the controller conforms to both protocols

Now that you know what you need to make the picker work, namely a delegate and a datasource, let's get back into Xcode and create them. Under **Classes** you have two files that need to be edited: `InstatwitViewController.h` and `InstatwitViewController.m`. Both files were created when you started the project.

The `.h` and `.m` files work together, with the header file (`.h`) declaring the class's interface, variable declarations, outlets, and actions, etc.; the implementation file (`.m`) holds the actual implementation code. We need to update the header file to state that our `InstatwitViewController` conforms to both the `UIPickerViewDataSource` and the `UIPickerViewDelegate` protocols.

Go ahead and add what's bolded.

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

@interface InstatwitViewController : UIViewController
<UIPickerViewDataSource, UIPickerViewDelegate> {
    NSArray* activities;
    NSArray* feelings;
}
@end
```

Here's where we say our class will conform to the `UIPickerViewDataSource` and `UIPickerViewDelegate` protocols.

We're going to set up two arrays for Mike: one for activities and one for feelings.



`InstatwitViewController.h`

Next, add Mike's activities and feelings to the implementation file

Now we're into `InstatwitViewController.m` file, the actual implementation. We'll need to add some methods to implement the required methods from the protocols, but we'll get back to that in a second. First, let's add the list from Mike. We're going to use the two arrays we declared in the header to store the words that Mike gave us.

All implementation code goes after `@implementation`. Here we indicate that we're realizing the `InstatwitViewController` interface we defined in the header.

```
#import "InstatwitViewController.h"
@implementation InstatwitViewController
```

The break here skips commented out default code that we're not using.



`InstatwitViewController.m`

Remove the `/*` marks that were here and then add the code. This method gets called on your view controller after the view is loaded from the `.xib` file. This is where you can do some initialization and setup for the view.

```
// Implement
viewDidLoad to do additional setup after loading the view,
typically from a nib.

- (void)viewDidLoad {
    [super viewDidLoad];

    activities = [[NSArray alloc] initWithObjects:@"sleeping",
    @"eating", @"working", @"thinking", @"crying", @"begging",
    @"leaving", @"shopping", @"hello worlding", nil];

    feelings = [[NSArray alloc] initWithObjects:@"awesome",
    @"sad", @"happy", @"ambivalent", @"nauseous", @"psyched",
    @"confused", @"hopeful", @"anxious", nil];
}
```

Here we establish the arrays with Mike's lists. We'll call them in a bit to fill in the picker.

The `@` before those strings tells the compiler to make them `NSString`s instead of `char*`. `NSString`s are real Objective-C classes, as opposed to a simple C-style character pointer. Most Objective-C classes use `NSString`s instead of `char*`s.

InstatwitViewController.m

```
-
(void)dealloc {
    [activities release];
    [feelings release];
    [super dealloc];
}

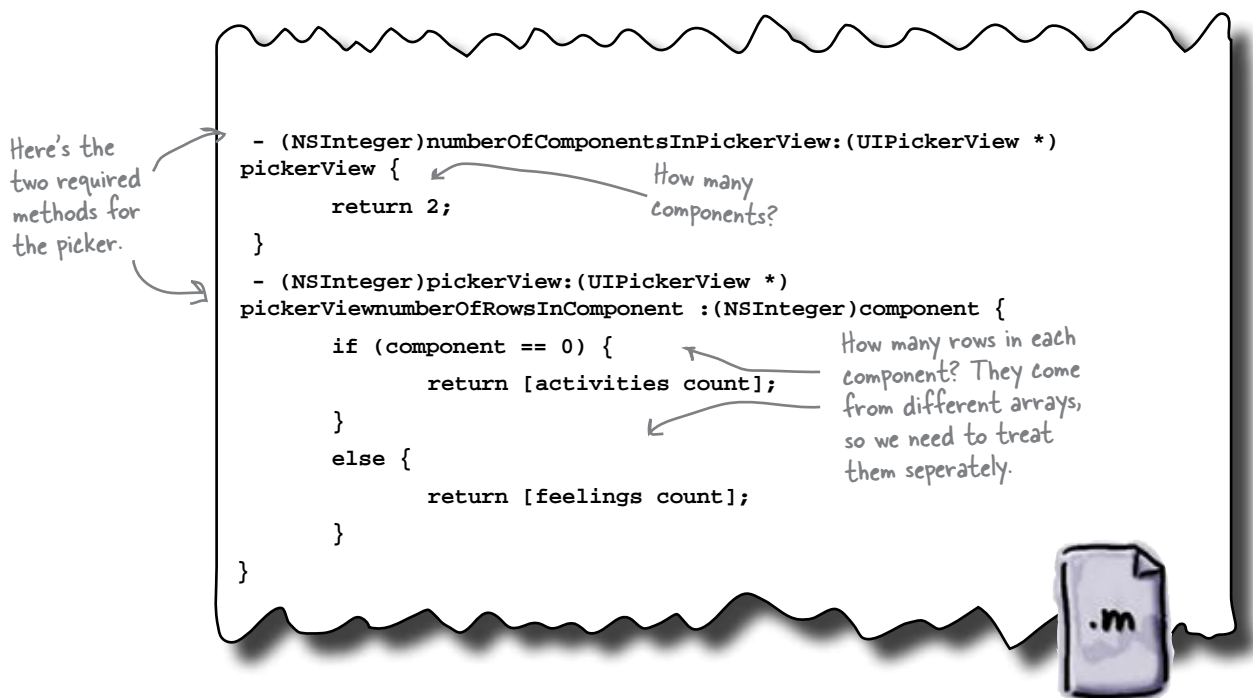
@end
```

You need to release all of these objects to clean up the memory, as an iPhone is small (so not much memory). We'll talk about memory a lot more in Chapter 3.

Now we just need the protocols...

The datasource protocol has two required methods

Let's focus on the datasource protocol methods first. We said in the header file that `InstatwitViewController` conforms to the `UIPickerViewDataSource` protocol. That protocol has two required methods, `numberOfComponentsInPickerView:pickerView` and `pickerView:numberOfRowsInComponent`. Since we know we want two wheels (components) in our view, we can start by putting that method in our implementation file:



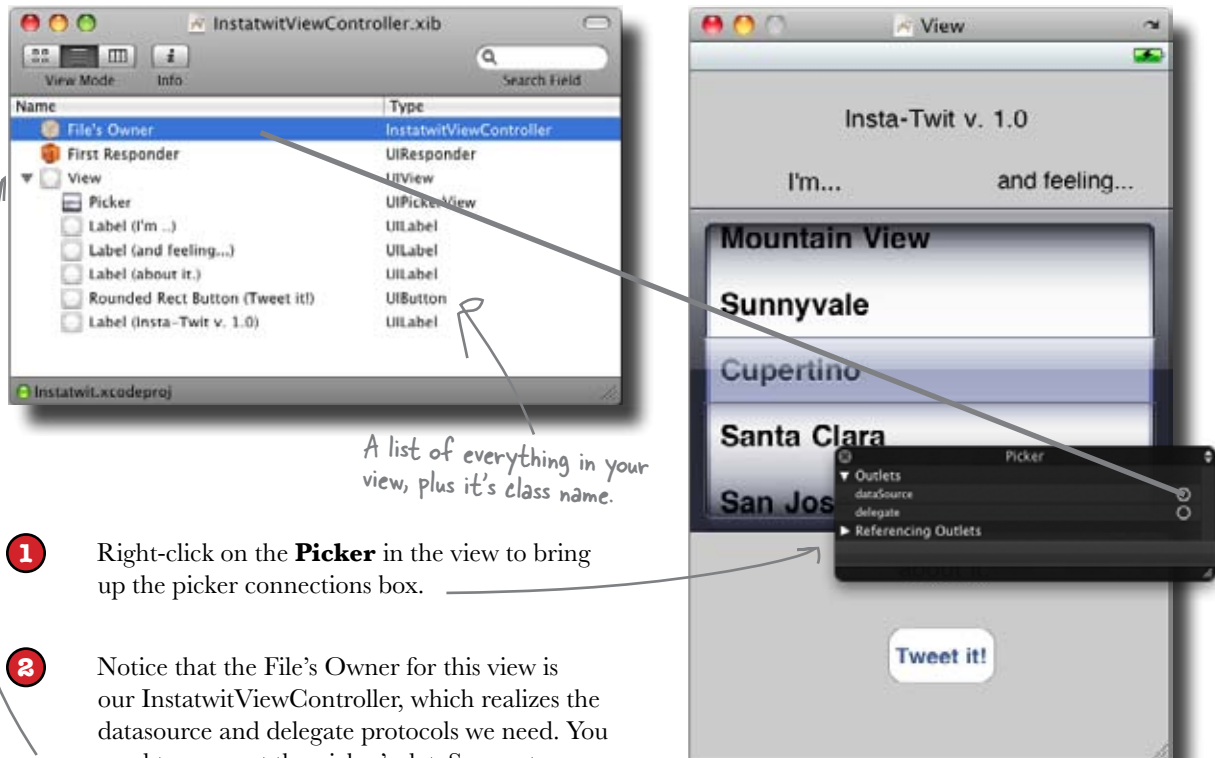
Our second method needs to return the number of rows for each component. The component argument will tell us which component the picker is asking about, with the first component (the activities) being component 0. The number of rows in each component is the just the number of items in the appropriate array.

InstatwitViewController.m

Now that we have the methods implemented, let's wire it up to the picker.

Connect the datasource just like actions and outlets

Now that the datasource protocol is implemented, the data is in place and it's just a matter of linking it to the picker. Hop back into Interface Builder to make that connection:



- 1 Right-click on the **Picker** in the view to bring up the picker connections box.
- 2 Notice that the File's Owner for this view is our `InstatwitViewController`, which realizes the datasource and delegate protocols we need. You need to connect the picker's `dataSource` to our controller, the File's Owner. To do that, click inside the circle next to the `dataSource`, and drag over the to File's Owner.



Watch it!

If you don't save in Interface Builder, it won't work!

Xcode will run the last saved version, not anything else.

On to the delegate...

There's just one method for the delegate protocol

The UIPickerViewDelegate protocol only has one required method (well, technically there are two optional methods, and you have to implement one of them). We're going to use `pickerView:titleForRow:forComponent:`. This method has to return an `NSString` with the title for the given row in the given component. Again, both of these values are indexed from 0, so we can use the component value to figure out which array to use, and then use the row value as an index.

The signature for these messages comes right out of the UIPickerViewDelegate and UIPickerViewDataSource documentation. Just cut and paste it if you want.

```
- (NSString *)pickerView:(UIPickerView *)pickerView
titleForRow:(NSInteger)row forComponent:(NSInteger)component {
    switch (component) {
        case 0:
            return [activities objectAtIndex:row];
        case 1:
            return [feelings objectAtIndex:row];
    }
    return nil;
}

- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}
```

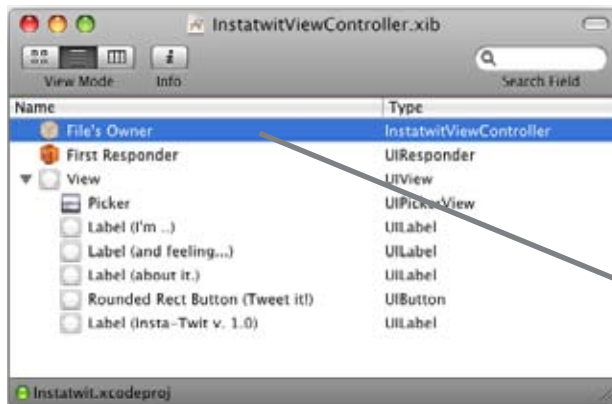
Return the string in the array at the appropriate location – row 0 is the first string, row 1 second, etc.

This gets called as your app is being shut down and the view is unloaded. We don't need it for now, so leave it as it was in the template.



InstatwitViewController.m

Now back to Interface Builder to wire up the delegate...



1 Right-click on the picker in the **Picker** again and bring up the connections window.

2 The File's Owner realizes the delegate protocol as well. Click inside the circle next to the delegate, and drag over the to File's Owner.



TEST DRIVE

Save your work in Interface Builder, go back into Xcode and save that, and Build and Run (⌘ return). When the Simulator pops up, you should see everything working!

Spin those dials – they're all the things on Mike's list and they work great!



there are no Dumb Questions

Q: What happens if I don't implement a required method in a protocol?

A: Your project will compile, but you'll get a warning. If you try to run your application, it will almost certainly crash with an "unrecognized selector" exception when a component tries to send your class the missing required message.

Q: What if I don't implement an optional method in a protocol?

A: That's fine. But whatever functionality that it would provide isn't going to be there. You do need to be a little careful in that sometimes Apple marks a couple of methods

as optional but you **have to implement at least one of them**. That's the case with the UIPickerViewDelegate. If you don't implement at least one of the methods specified in the docs, your app will crash with an error when you try to run it.

Q: Are there limits to the number of protocols a class can realize?

A: Nope. Now, the more you realize, the more code you're going to need to put in that class, so there's a point where you really need to split things off into different classes to keep the code manageable. But technically speaking, you can realize as many as you want.

Q: I'm still a little fuzzy, what's the difference between the interface we put in a header file and a protocol?

A: An interface in a header file is how Objective-C declares the properties, fields, and messages a class responds to. It's like a header file in C++ or the method declarations in a Java file. However, you have to provide implementation for everything in your class's interface. A protocol on the other hand is just a list of messages—there is no implementation. It's the class that realizes the protocol that has to provide implementation. These are equivalent to interfaces in Java and pure virtual methods in C++.



BULLET POINTS

- The picker needs a delegate and a data-source to work.
- In a picker, each dial is a component.
- In a picker, each item is a row.
- Protocols define the messages your class must realize—some of them might be optional.

OK, that's great and all. It looks really nice. But the "Tweet it!" button doesn't do anything yet...

Now let's get that button talking to Twitter...

We got the picker working, but if you try out the "Tweet it!" button, nothing happens when something's selected. We still need to get the button responding to Mike and then get the whole thing to talk to Twitter.

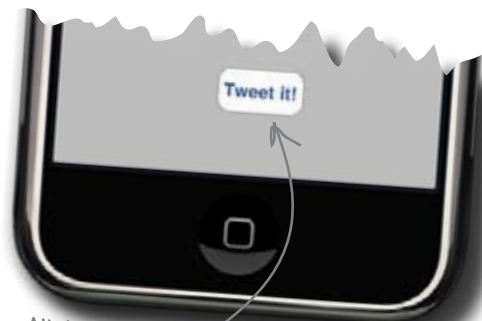


BRAIN BARBELL

Think about what we need to do to get the button working. What files will we use? What will the button actually do?

The button needs to be connected to an event

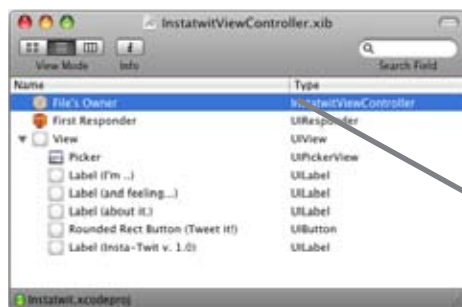
We need to wire up the button like we did in Chapter 1. Once Mike has selected what he's doing and feeling, he'll hit **"Tweet it!"** Then we need to get his selections out of the picker and send them to Twitter.



All that, in one little button...



So we just need to go back to IB and wire up the TouchUpInside event again, right?



Yes, but what will we wire that event to?

Without an action, your button won't work!

We learned about actions in Chapter 1, and without one there won't be anything in the connections window to wire up in Interface Builder.

Here's the action we created for the button press in Chapter 1:

`- (IBAction) buttonPressed:(id)sender;`

IB = Interface Builder

This is the name of the method that will get called. The name can be anything, but the method must have one argument of type (id).

All IBAction messages take one argument: the sender of the message. This is the element that triggered the action.



Exercise

We need to change both the header and implementation files for the InstatwitViewController.

- 1 Start with the header and add an IBAction named `sendButtonTapped`.
- 2 Then provide an implementation for that method in our `.m` file, and write a message to the log so you know it worked before sending to Twitter



Exercise Solution

Declare your IBAction in the header file and provide the implementation in the .m file.

1

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

@interface InstatwitViewController : UIViewController
<UIPickerViewDataSource, UIPickerViewDelegate> {
    NSArray* activities;
    NSArray* feelings;
}
- (IBAction) sendButtonTapped: (id) sender;
```

The IBAction is what allows the code to respond to a user event, remember...

Declare your IBAction here so we can use it in the .m file and Interface Builder knows we have an action available.



InstatwitViewController.h

2

```
- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];
    // Release any cached data, images, etc that aren't in use.
}

- (IBAction) sendButtonTapped: (id) sender {
    NSLog(@"Tweet button tapped!");
}
```

Same method declaration as the .h

This will give you the output on the console..



InstatwitViewController.m

Now go back and hook it up with Interface Builder...



Test Drive

Save, then Build and Run. You should get the “Tweet button tapped!” message in the console.



So now we need to get the data from that picker, right? Would an IBOutlet be the right thing for that?

Yes! An IBOutlet provides a reference to the picker.

In Chapter 1, we used an outlet to access and change the text field value on the button. Now, to gather up the actual message to send to Twitter, we need to extract the values chosen from the picker, then create a string including the label text.

So far the picker has been calling us when it needed information; this time, when Mike hits the “Tweet it” button, we need to get data out of the picker. We’ll use an IBOutlet to do that.



Add the IBOutlet and property to our view controller

In addition to declaring the IBOutlet, we'll declare a property with the same name. We'll talk more about properties in the next chapter, but in short, that will get us proper memory management and let the Cocoa Touch framework set our tweetPicker field when our nib loads.

Start with the header file...

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

@interface InstatwitViewController : UIViewController
<UIPickerViewDataSource, UIPickerViewDelegate> {

    IBOutlet UIPickerView *tweetPicker;
    NSArray* activities;
    NSArray* feelings;
}

@property (nonatomic, retain) UIPickerView* tweetPicker;
-(IBAction) sendButtonTapped: (id) sender;

@end
```

Here we declare a field in the class called tweetPicker. The type is a pointer to a UIPickerView.

Here's our outlet declaration. This lets Interface Builder know you have something to connect to. IBOutlets are actually #defined to nothing; they're just there for Interface Builder.

The property for tweetPicker has some memory management options that we'll explain more in Chapter 3.



InstatwitViewController.h

...and then add the implementation.

```
#import "InstatwitViewController.h"
@implementation InstatwitViewController
@synthesize tweetPicker;
```



InstatwitViewController.m

@synthesize goes along with the @property declaration in the .h file. See Chapter 3 for more info...

```
- (void)dealloc {
    [tweetPicker release];
    [activities release];
    [feelings release];
    [super dealloc];
}
```

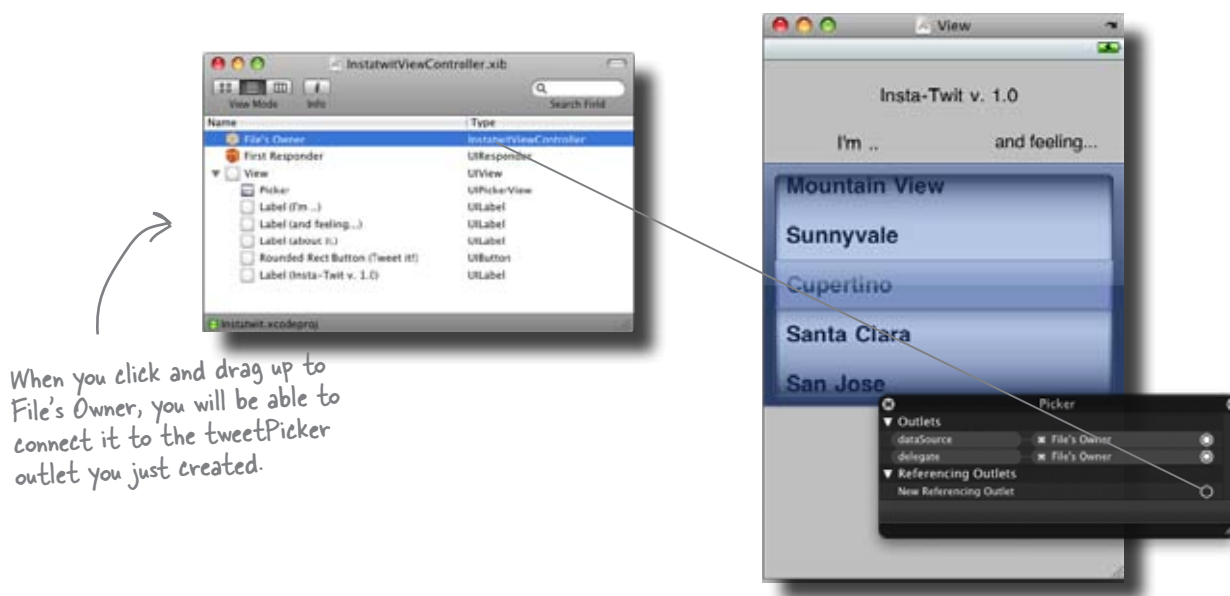
```
@end
```

The last thing you need to do with tweetPicker is release our reference to it – another memory thing. We'll come back to the memory management in Chapter 3, we promise.

What's next?

Connect the picker to our outlet

You're probably expecting this by now! Back into Interface Builder to make the connection from the UIPickerView to the IBOutlet in our view controller. Right-click on the UIPickerView, grab the circle next to the “New Referencing Outlet,” and drop it on File’s Owner—our InstatwitViewController sporting its new tweetPicker outlet.



What do you need to do now to get the data out of the picker and into your Twitter message? Think about the “Tweet it!” button press action and how that will need to change...

Use our picker reference to pull the selected values

Now all that's left is to use our reference to the picker to get the actual values Mike selects. We need to reimplement the `sendButtonTapped` method to pull the values from the picker. Looking at the `UIPickerView` documentation, the method we need is `selectedRowInComponent:`. That method returns a row value, which, just like before, we can use as an index into our arrays.

Here's the implementation for our callback. We need to create a string and fill in the values from the picker. the "%@" in the string format get replaced with the values we pass in.

To figure out what Mike chose on the picker, we need to ask the picker what row is selected, and get the corresponding string from our arrays.

```
- (IBAction) sendButtonTapped: (id) sender {
    NSString* themessage = [NSString stringWithFormat:@"I'm %@ and feeling %@
    about it."],

        [activities objectAtIndex:[tweetPicker selectedRowInComponent:0]],
        [feelings objectAtIndex:[tweetPicker selectedRowInComponent:1]]];

    NSLog(themessage);
    NSLog(@"Tweet button tapped!");
}
```

Pull this log message out and put in one to see what the final Twitter message will be.

We want to build a new string with the full tweet text in it, so we'll use `NSString`'s `stringWithFormat` method to create a templated string. There are lots of other options you could use with a string format, like characters, integers, etc., but for now we just need to insert the two selected strings, so we'll use "%@".



InstatwitViewController.m

We're just going to log this message to the console so we can see the string we're building, and then we'll send this to Twitter in just a minute. Let's make sure we implemented this correctly first before tweeting to the whole world...



Test Drive

OK, try it out. You should get a convincing tweet in the console:



Once we add the Twitter info, this is what will actually show up in your feed as a tweet.

All that's left is to talk to Twitter—we'll help you with that.



Ready Bake Code

To post to Twitter, we're going to use their API. Rather than go into a Twitter API tutorial, we'll give you the code you need to tweet the string. Type the code you see below into the `InstatwitViewController.m`, just below the `NSLog` with the Twitter message in the `sendButtonTapped` method.

```
//TWITTER BLACK MAGIC
    NSMutableURLRequest *theRequest=[NSMutableURLRequest requestWithURL:[NSURL
URLWithString:@"http://YOUR_TWITTER_USERNAME:YOUR_TWITTER_PASSWORD@twitter.com/
statuses/update.xml"]

        cachePolicy:NSURLRequestUseProtocolCachePolicy
        timeoutInterval:60.0];

    [theRequest setHTTPMethod:@"POST"];
    [theRequest setHTTPBody:[NSString stringWithFormat:@"status=%@",
thessage] dataUsingEncoding:NSUTF8StringEncoding]];

    NSURLResponse* response;
    NSError* error;

    NSData* result = [NSURLConnection sendSynchronousRequest:theRequest
returningResponse:&response error:&error];

    NSLog(@"%@", [[[NSString alloc] initWithData:result
encoding:NSUTF8StringEncoding] autorelease]);
//      END TWITTER BLACK MAGIC
```

Your username and password
need to go in here.



`InstatwitViewController.m`

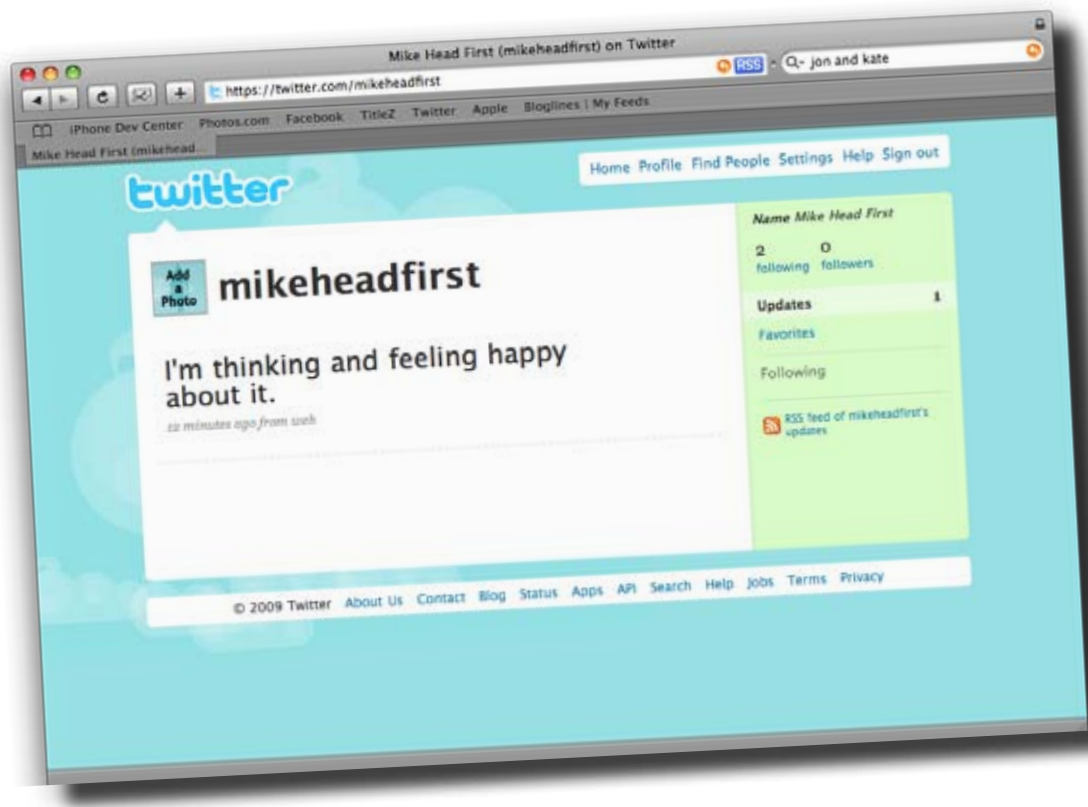


**If you don't have a Twitter account,
just go get one!**

Just go to twitter.com and register. Once you do that, you can enter your username and password, and this will work like a charm.

After adding that code, you can just save, build and go. It will now show up on your Twitter feed. Go ahead, try it out!

mike's feeling great about your app



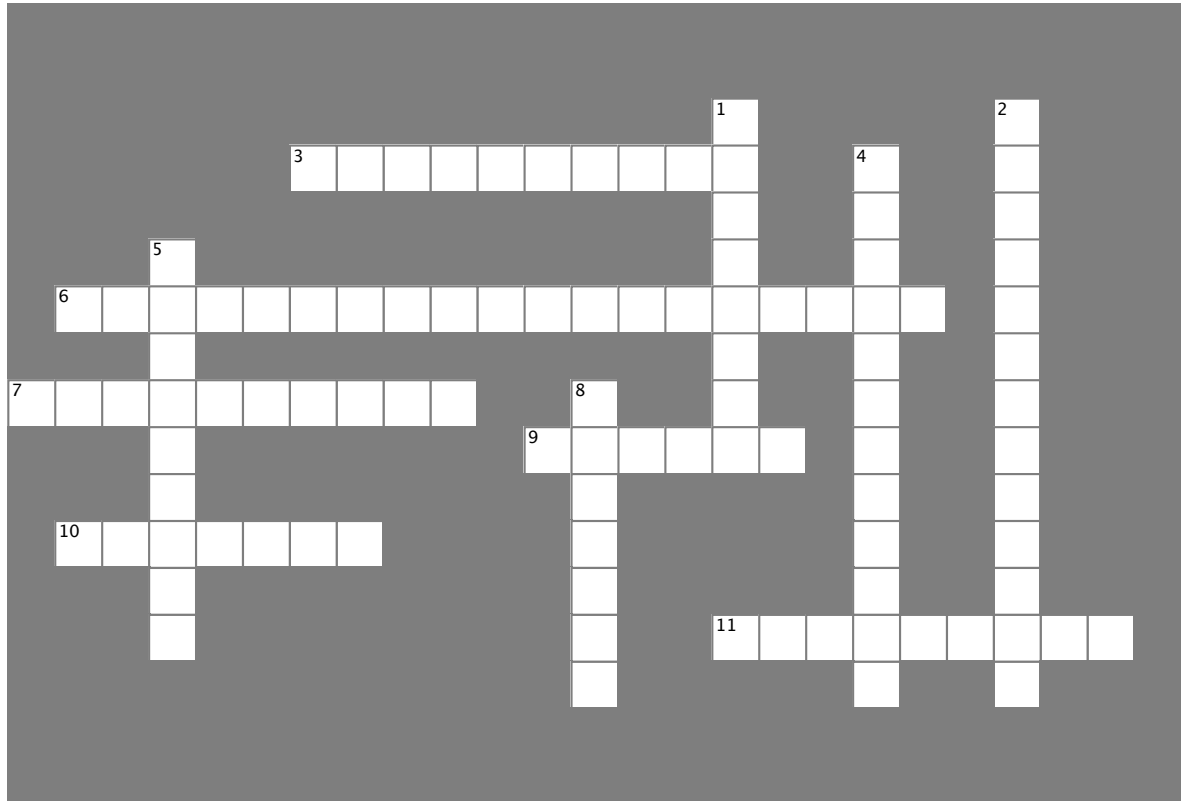
That is great! Now, Renee is happy and feels included and I don't actually have to talk out loud about my feelings. At all. Ever.





iPhonecross

Flex your vocab skills with this crossword.



Across

3. This typically handles the information itself in the app.
6. This is the document Apple uses to evaluate apps for the App Store.
7. You see this listed in the view and it controls the view.
9. This component allows for controlled input from several selections.
10. This type of app is typically one screen, and gives you the basics with minimal interaction.
11. These define to which messages the datasource and delegate respond.

Down

1. This typically contains the logic that controls the flow of information in an app.
2. The best way to figure out what protocols you need to worry about is to check the _____.
4. This app type typically involves hierarchical data.
5. This app type is mostly custom controllers and graphics.
8. The other name for an *.xib file.



Exercise

We've listed a couple of descriptions of some different apps. Using the app description, sketch out a rough view and answer the questions about each one.

1

Generic giant button app

There are several of these currently up for sale on the app store. This app consists of pushing a big button and getting some noise out of your iPhone.

What type of app is this?

.....

What are the main concerns in the HIG about this app type?

.....

.....

.....



2

Book inventory app

This app's mission is to keep a list of the books in your library, along with a quick blurb of what it's about and the author.

What type of app is this?

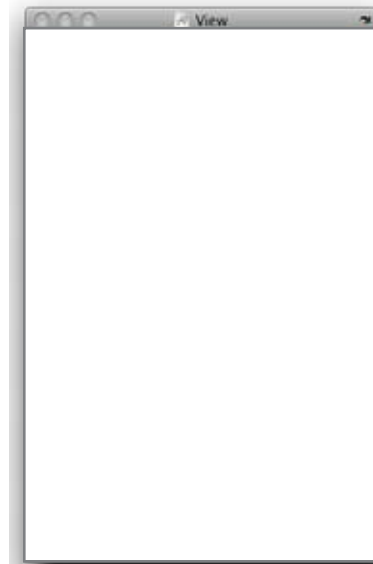
.....

What are the main concerns in the HIG about this app type?

.....

.....

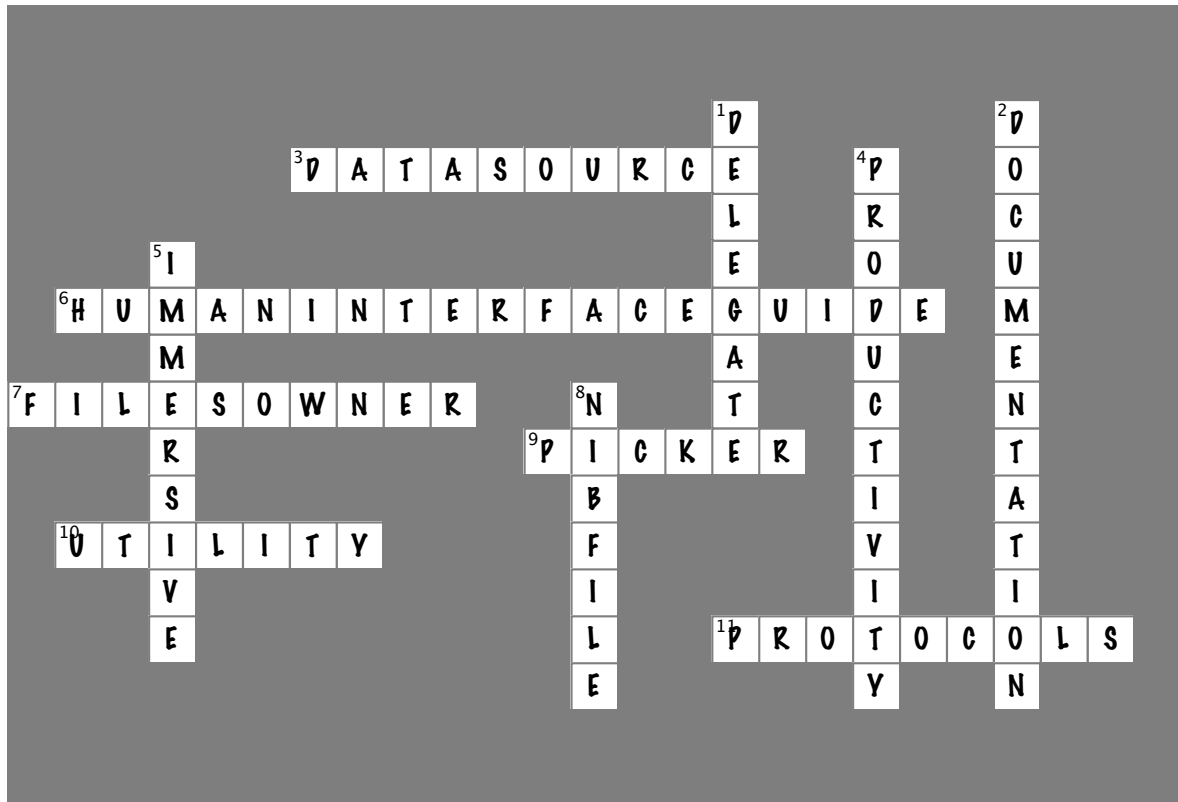
.....





iPhonecross Solution

Flex your vocab skills with this crossword.



Across

3. This typically handles the information itself in the app. [DATASOURCE]
6. This is the document apple uses to evaluate apps for the App Store. [HUMANINTERFACEGUIDE]
7. You see this listed in the view and it controls the view. [FILEOWNER]
9. This component allows for controlled input from several selections. [PICKER]
10. This type of app is typically one screen, and gives you the basics with minimal interaction. [UTILITY]
11. These define to which messages the datasource and delegate respond. [PROTOCOLS]

Down

1. This typically contains the logic that controls the flow of information in an app. [DELEGATE]
2. The best way to figure out what protocols you need to worry about is to check the _____. [DOCUMENTATION]
4. This app type typically involves hierarchical data. [PRODUCTIVITY]
5. This app type is mostly custom controllers and graphics. [IMMERSIVE]
8. The other name for an *.xib file. [NIBFILE]



Exercise Solution

1

Generic giant button app

There are several of these currently up for sale on the app store. This app consists of pushing a big button and getting some noise out of your iPhone.

What type of app is this?

An immersive app

What are the main concerns in the HIG about this app type?

The big thing Apple cares about is that controls "provide an internally consistent experience." So everything can be custom, it needs to be focused and well organized.

2

Book inventory app.

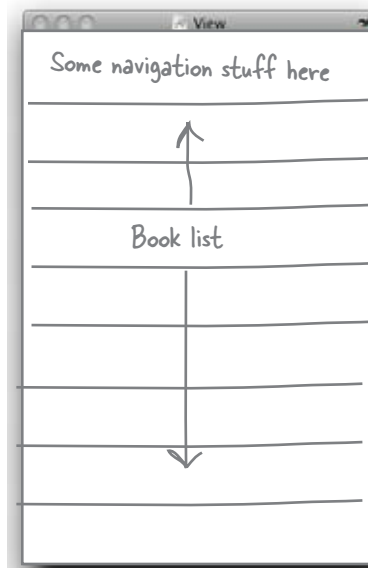
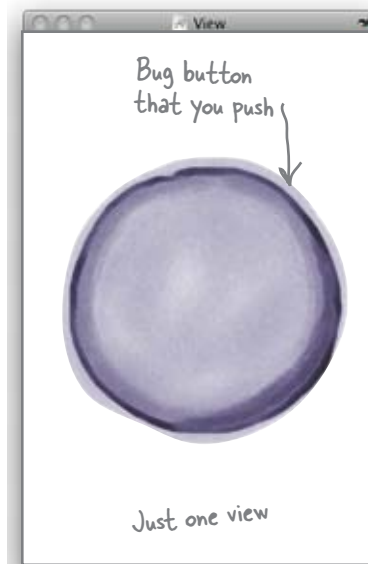
This app's mission is to keep a list of the books in your library, along with a quick blurb of what it's about and the author.

What type of app is this?

A productivity app

What are the main concerns in the HIG about this app type?

The HIG has many more specific rules about this app type, because you'll be using standard controls. EACH control needs to be checked out for proper usage.



Another view for details, need to figure out how to get to it...



Your iPhone Toolbox

You've got Chapter 2 under your belt and now you've added protocols, delegates, and datasources to your toolbox. For a complete list of tooltips in the book, go to <http://www.headfirstlabs.com/iphonedev>.

Protocols

Define the messages your datasource and delegate must respond to.
Are declared in the header (.h) file.
Some of them might be optional.

Delegate

Responsible for the behavior of a UI element.

Contains the logic that controls the flow of information, like saving or displaying data, and which view is seen when.

Can be in same object as the datasource, but has its own specific protocols.

Datasource

Provides the bridge between the control and the data it needs to show.

Works with databases, plists, images, and other general info that your app will need to display.

Can be the same object as a delegate, but has its own specific protocols.



BULLET POINTS

- The picker needs a delegate and datasource to work.
- In a picker, each dial is a component.
- In a picker, each item is a row.
- Protocols define the messages your class must realize—some of them might be optional.

This is Renee, Mike's girlfriend



It's so great that Mike and I are communicating now! But I've noticed that Mike's starting to sound like he's in a rut, saying the same thing over and over again! Is there something we need to talk about?

Sounds like Mike is going to need some modifications to InstaTwit to keep his relationship on solid ground...