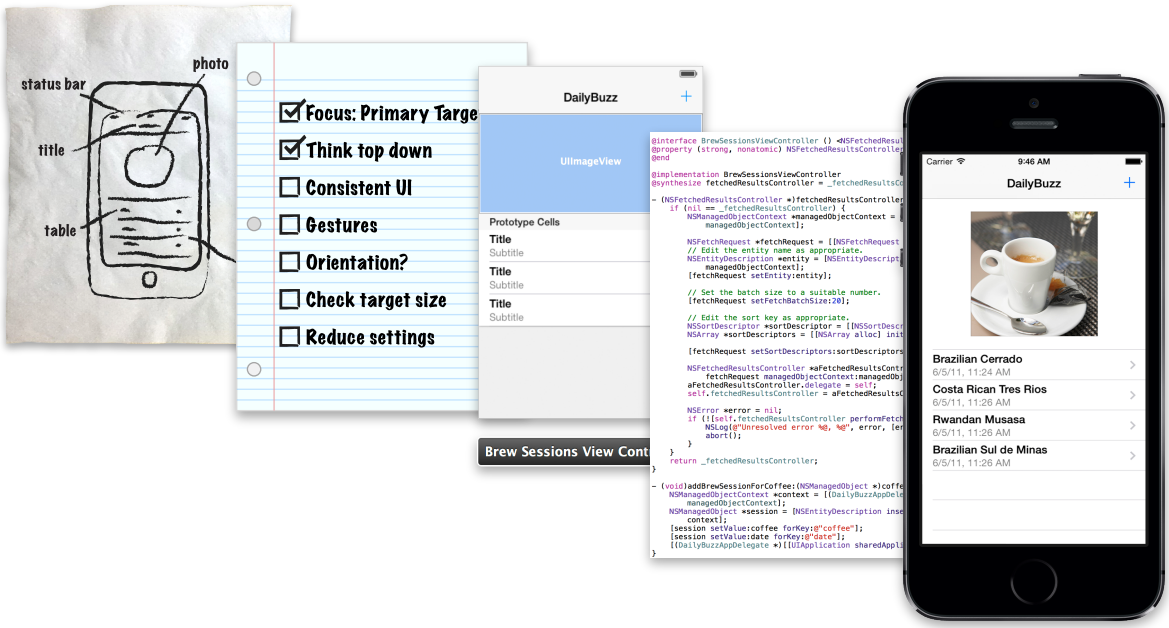
Structuring an App

App Development Process

Although the task of developing an app may seem daunting, the process can be distilled into several digestible steps. The steps that follow will help you get started and guide you in the right direction as you develop your first app.



Defining the Concept

Every great app starts with a concept.

The best way to arrive at that concept is to consider the problem you want your app to solve. Great apps solve a single, well-defined problem. For example, the Settings app allows users to adjust all of the settings on their device. It provides a single interface for users to accomplish a related set of tasks.

Here are some key questions to consider when arriving at a concept:

Who is your audience? Your app content and experience will differ depending on whether you're writing a children's game, a to-do list app, or even a test app for your own learning.

What is the purpose of your app? It's important for an app to have a clearly defined purpose. Part of defining the purpose is understanding what one thing will motivate users to use your app.

What problem is your app trying to solve? An app should solve a single problem well instead of trying to offer solutions to multiple distinct problems. If you find that your app is trying to solve unrelated problems, you might consider writing multiple apps.

What content will your app incorporate? Consider what type of content your app will present to users and how they'll interact with it. Design the user interface to complement the type of content that's presented in the app.

An app concept doesn't have to be completely polished or finished when you start developing your app. Still, it helps to have an idea of where you're going and what you need to do to get there.

Designing a User Interface

After you have a concept for your app, designing a good user interface is the next step to creating a successful app. A user needs to be able to interact with the app interface in the simplest way possible. Design the interface with the user in mind, and make it efficient, clear, and straightforward.

Perhaps the most challenging thing about building a user interface is translating your concept into a design and then implementing that design. To help simplify this process, use storyboards. **Storyboards** let you design and implement your interface in a single step using a graphical environment. You can see exactly what you're building while you're building it, get immediate feedback about what's working and what's not, and make instantly visible changes to your interface.

When you build an interface in a storyboard, you're working with views. **Views** display content to the user. In "Tutorial: Basics" (page 8), you began to define a user interface for the ToDoList app using a single view in a storyboard scene. As you develop more complex apps, you'll create interfaces with more scenes and more views.

In "Tutorial: Storyboards" (page 47), you'll finish building the user interface for your ToDoList app using several different views to display different types of content. You'll learn more about working with views and storyboards to design and create a user interface in "Designing a User Interface" (page 36).

Defining the Interaction

A user interface doesn't do much without any logic backing it. After you've created an interface, you define how users can interact with what they see by writing code to respond to user actions in your interface.

Before you start thinking about adding the behaviors for your interface, it's important to understand that iOS apps are based on event-driven programming. In **event-driven programming**, the flow of the app is determined by events: system events or user actions. The user performs actions on the interface, which trigger events in the app. These events result in the execution of the app's logic and manipulation of its data. The app's response to user action is then reflected back in the interface.

As you define how a user can interact with your interface, keep event-driven programming in mind. Because the user, rather than the developer, is in control of when certain pieces of the app code get executed, you want to identify exactly which actions a user can perform and what happens in response to those actions.

You define much of your event-handling logic in **view controllers**. You'll learn more about working with view controllers in "Defining the Interaction" (page 42). Afterward, you'll apply these concepts to add functionality and interactivity to your ToDoList app in "Tutorial: Storyboards" (page 47).

Implementing the Behavior

After you've defined the actions a user can perform in your app, you implement the behavior by writing code.

When you write code for iOS apps, most of your time is spent working with the **Objective-C programming language**. You'll learn more about Objective-C in the third module, but for now, it helps to have some basic familiarity with the vocabulary of the language.

Objective-C is built on top of the C programming language and provides object-oriented capabilities and a dynamic runtime. You get all of the familiar elements, such as primitive types (`int`, `float`, and so on), structures, functions, pointers, and control flow constructs (`while`, `if...else`, and `for` statements). You also have access to the standard C library routines, such as those declared in `stdlib.h` and `stdio.h`.

Objects Are Building Blocks for Apps

When you build an iOS app, most of your time is spent working with objects.

Objects package data with related behavior. You can think of an app as a large ecosystem of interconnected objects that communicate with each other to solve specific problems, such as displaying a visual interface, responding to user input, or storing information. You use many different types of objects to build your app, ranging from interface elements, such as buttons and labels, to data objects, such as strings and arrays.

Classes Are Blueprints for Objects

A **class** describes the behavior and properties common to any particular type of object.

In the same way that multiple buildings constructed from the same blueprint are identical in structure, every instance of a class shares the same properties and behavior as all other instances of that class. You can write your own classes or use framework classes that have been defined for you.

You make an object by creating an **instance** of a particular class. You do this by allocating it and initializing it with acceptable default values. When you **allocate** an object, you set aside enough memory for the object and set all instance variables to zero. **Initialization** sets an object's initial state—that is, its instance variables and properties—to reasonable values and then returns the object. The purpose of initialization is to return a usable object. You need to both allocate and initialize an object to be able to use it.

One of the fundamental concepts in Objective-C programming is **class inheritance**, the idea that a class inherits behaviors from a parent class. When one class inherits from another, the child—or **subclass**—inherits all the behavior and properties defined by the parent. The subclass can define its own additional behavior and properties or override the behavior of the parent. This gives you the ability to extend the behaviors of a class without duplicating its existing behavior.

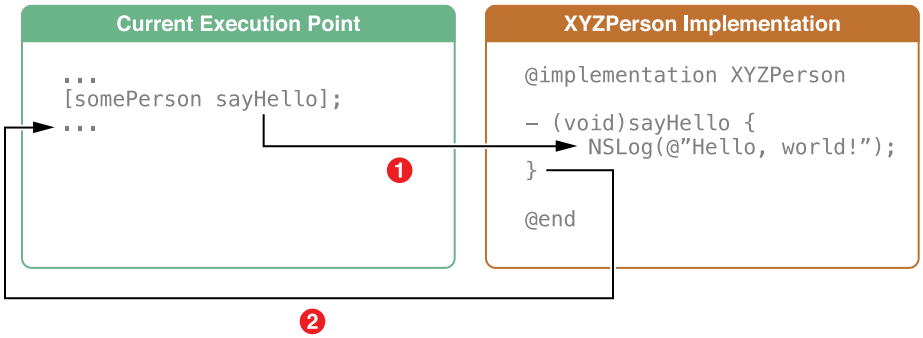
Objects Communicate Through Messages

Objects interact by sending each other messages at runtime. In Objective-C terms, one object **sends a message** to another object by **calling a method** on that object.

Although there are several ways to send messages between objects in Objective-C, by far the most common is the basic syntax that uses square brackets. If you have an object `somePerson` of class `Person`, you can send it the `sayHello` message like this:

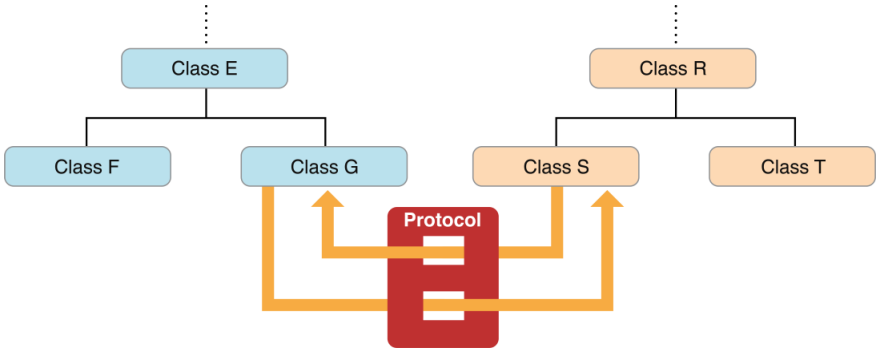
```
[somePerson sayHello];
```

The reference on the left, `somePerson`, is the receiver of the message. The message on the right, `sayHello`, is the name of the method to call on that receiver. In other words, when the above line of code is executed, `somePerson` will be sent the `sayHello` message.



Protocols Define Messaging Contracts

A **protocol** defines a set of behavior that's expected of an object in a given situation. A protocol comes in the form of a programmatic interface, one that any class may choose to implement. Using protocols, two classes distantly related by inheritance can communicate with each other to accomplish a certain goal, such as parsing XML code or copying an object.



Any class that can provide behavior that's useful to other classes can declare a programmatic interface for vending that behavior anonymously. Any other class can choose to adopt the protocol and implement one or more of its methods, making use of the behavior.

Incorporating the Data

After you implement your app's behavior, you create a **data model** to support your app's interface. An app's data model defines the way you maintain data in your app. Data models can range from a basic dictionary of objects to complex databases.

Your app's data model should reflect the app's content and purpose. There should be a clear correlation between the interface and the data, even though the user doesn't interact with the data directly.

A good data model is essential to creating a solid foundation for your app. It makes it easier to build a scalable app, improve functionality, and make changes to your features. You'll learn more about defining your own data model in "Incorporating the Data" (page 70).

Use the Right Resources

Design patterns are best practices for solving common problems in apps. Use design patterns to help you define the structure of your data model and its interaction with the rest of your app. When you understand and use the right design patterns, you can more easily create an app that's simple and efficient. You'll learn more about design patterns in "Using Design Patterns" (page 72).

As you start implementing your model, remember that you don't have to implement everything from scratch. There are a number of **frameworks** that provide existing functionality for you to build upon. For instance, the **Foundation framework** includes classes representing basic data types—such as strings and numbers—as well as collection classes for storing other objects. It's recommended that, where possible, you use existing framework classes—or subclass them to add your own app's features—instead of trying to reimplement their functionality. In this way, you can create an efficient, functional, sophisticated app. You'll learn more about the capabilities of the Foundation framework in "Working with Foundation" (page 75).

Often, you'll write your own **custom classes** as part of your data model. Writing a custom class gives you control over how you organize the internal structure of your app. You'll learn about creating custom classes in "Writing a Custom Class" (page 86).

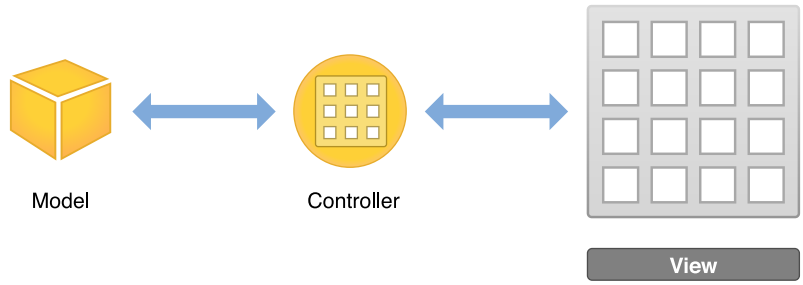
Incorporate Real Data

When you first test your data model, you may want to use static or fake data. This way, you don't have to worry about supplying real data until you know the model is assembled and connected properly. After you've defined a data model that's working properly, you can pull real data into your app.

The remainder of this guide takes you through these steps in more detail. As you make your way through the app development process, you'll learn the necessary conceptual material and then put it to use in the tutorials.

Designing a User Interface

Views are the building blocks for constructing your user interface. It's important to understand how to use views to present your content in a clear, elegant, and useful way. Creating a great user interface that effectively showcases your app's content is essential to building a successful app. In this chapter, you'll learn about creating and managing views in a storyboard to define your interface.



The View Hierarchy

Views not only display themselves onscreen and react to user input, they also serve as containers for other views. As a result, views in an app are arranged in a hierarchical structure called the view hierarchy. The **view hierarchy** defines the layout of views relative to other views. Within that hierarchy, view instances enclosed within a view are called **subviews**, and the parent view that encloses a view is referred to as its **superview**. Even though a view instance can have multiple subviews, it can have only one superview.

At the top of the view hierarchy is the **window** object. Represented by an instance of the `UIWindow` class, a window serves as the basic container into which you can add your view objects for display onscreen. By itself, a window doesn't display any content. To display content, you add a **content view** (with its hierarchy of subviews) to the window.

For a content view and its subviews to be visible to the user, the content view must be inserted into a window's view hierarchy. When you use a storyboard, this placement is configured automatically for you. The application object loads the storyboard, creates instances of the relevant view controller classes, unarchives the content view hierarchies for each view controller, and then adds the content view of the initial view controller into the window. You'll learn more about managing view controllers in the next chapter; for now, you'll focus on creating a hierarchy within a single view controller in your storyboard.

Building an Interface Using Views

When you design your app, it's important to know what kind of view to use for what purpose. For example, the kind of view you use to gather input text from a user, such as a text field, is different from what you might use to display static text, such as a label. Apps that use UIKit views for drawing are easy to create because you can assemble a basic interface quickly. A UIKit view object is an instance of the `UIView` class or one of its subclasses. The UIKit framework provides many types of views to help present and organize data.

Although each view has its own specific function, UIKit views can be grouped into seven general categories:

Category	Purpose	Examples
 Content	Display a particular type of content, such as an image or text.	Image view, label
 Collections	Display collections or groups of views.	Collection view, table view
 Controls	Perform actions or display information.	Button, slider, switch
 Bars	Navigate, or perform actions.	Toolbar, navigation bar, tab bar
 Input	Receive user input text.	Search bar, text view
 Containers	Serve as containers for other views.	View, scroll view
Modal	Interrupt the regular flow of the app to allow a user perform some kind of action.	Action sheet, alert view

You can assemble views graphically using Interface Builder. **Interface Builder** provides a library of the standard views, controls, and other objects that you need to build your interface. After dragging these objects from the library, you drop them onto the canvas and arrange them in any way you want. Next, use inspectors to configure those objects before saving them in a storyboard. You see the results immediately, without the need to write code, build, and run your app.

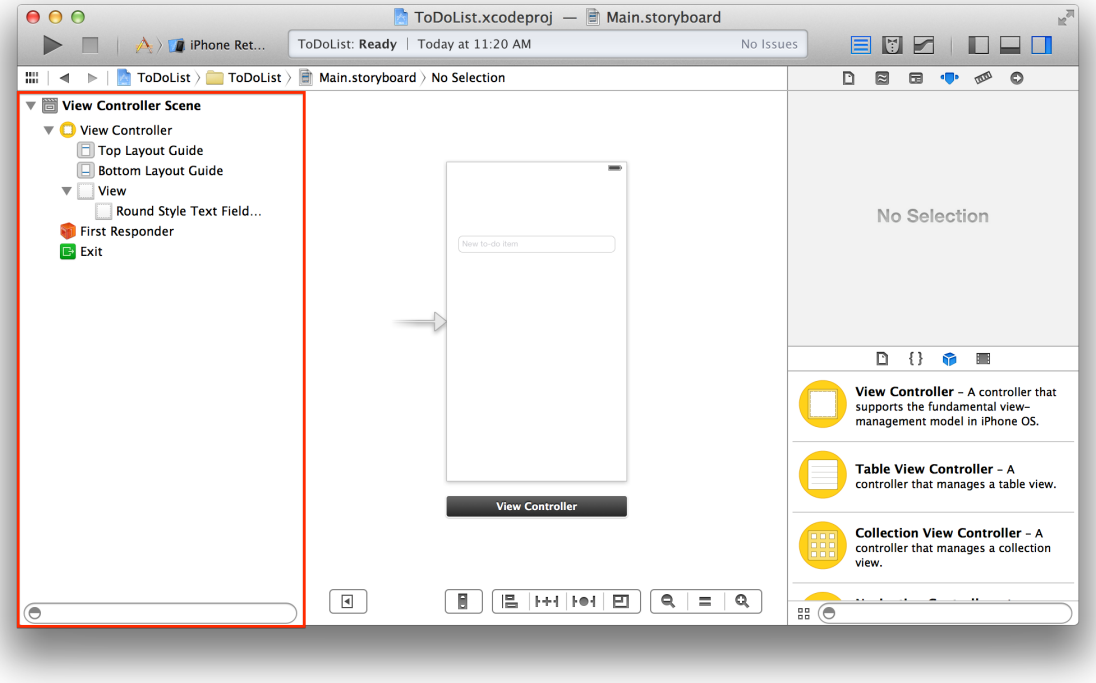
The UIKit framework provides standard views for presenting many types of content, but you can also define your own custom views by subclassing `UIView` (or its descendants). A custom view is a subclass of `UIView` in which you handle all of the drawing and event-handling tasks yourself. You won't be using custom views in these tutorials, but you can learn more about implementing a custom view in "Defining a Custom View" in *View Programming Guide for iOS*.

Use Storyboards to Lay Out Views

You use a storyboard to lay out your hierarchy of views in a graphical environment. Storyboards provide a direct, visual way to work with views and build your interface.

As you saw in the first tutorial, storyboards are composed of scenes, and each scene has an associated view hierarchy. You drag a view out of the object library and place it in a storyboard scene to add it automatically to that scene's view hierarchy. The view's location within that hierarchy is determined by where you place it. After you add a view to your scene, you can resize, manipulate, configure, and move it on the canvas.

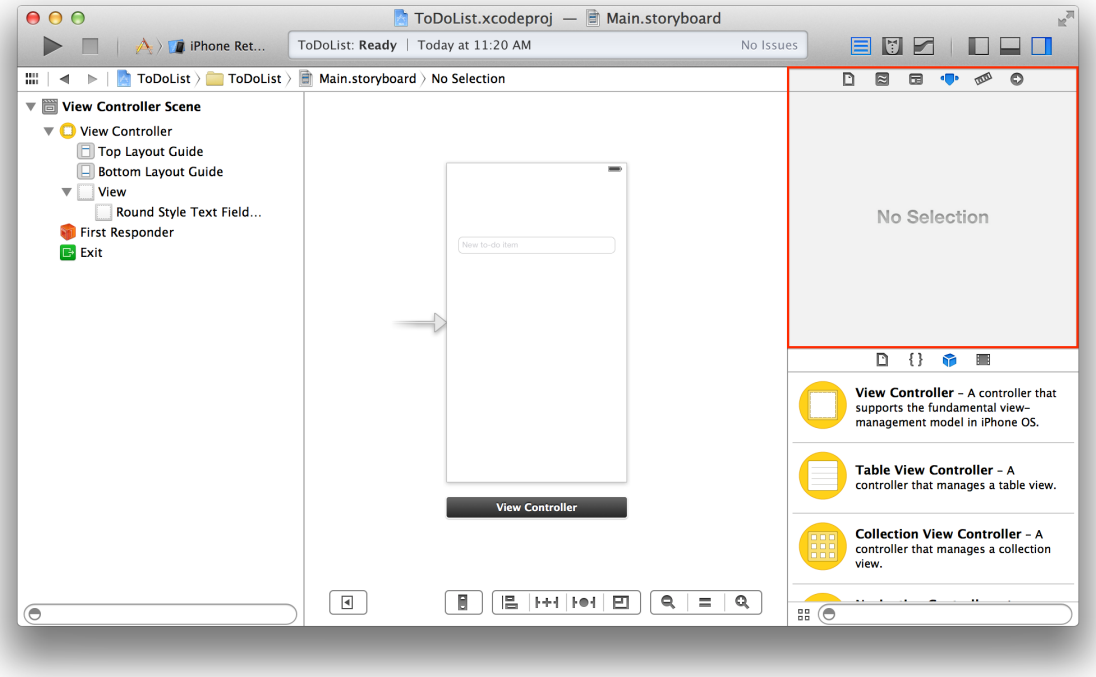
The canvas also shows an outline view of the objects in your interface. The **outline view**—which appears on the left side of the canvas—lets you see a hierarchical representation of the objects in your storyboard.



The view hierarchy that you create graphically in a storyboard scene is effectively a “shrinkwrapped” set of Objective-C objects. At runtime, these shrinkwrapped objects are unarchived. The result is a hierarchy of instances of the relevant classes configured with the properties you’ve set visually using the various inspectors in the utility area.

Use Inspectors to Configure Views

When working with views in a storyboard, the **inspector pane** is an essential tool. The inspector pane appears in the utility area above the Object library.



Each of the inspectors provides important configuration options for elements in your interface. When you select an object, such as a view, in your storyboard, you can use each of the inspectors to customize different properties of that object.

- **File.** Lets you specify general information about the storyboard.
- **Quick Help.** Provides useful documentation about an object.
- **Identity.** Lets you specify a custom class for your object and define its accessibility attributes.
- **Attributes.** Lets you customize visual attributes of an object.
- **Size.** Lets you specify an object's size and Auto Layout attributes.
- **Connections.** Lets you create connections between your interface and source code.

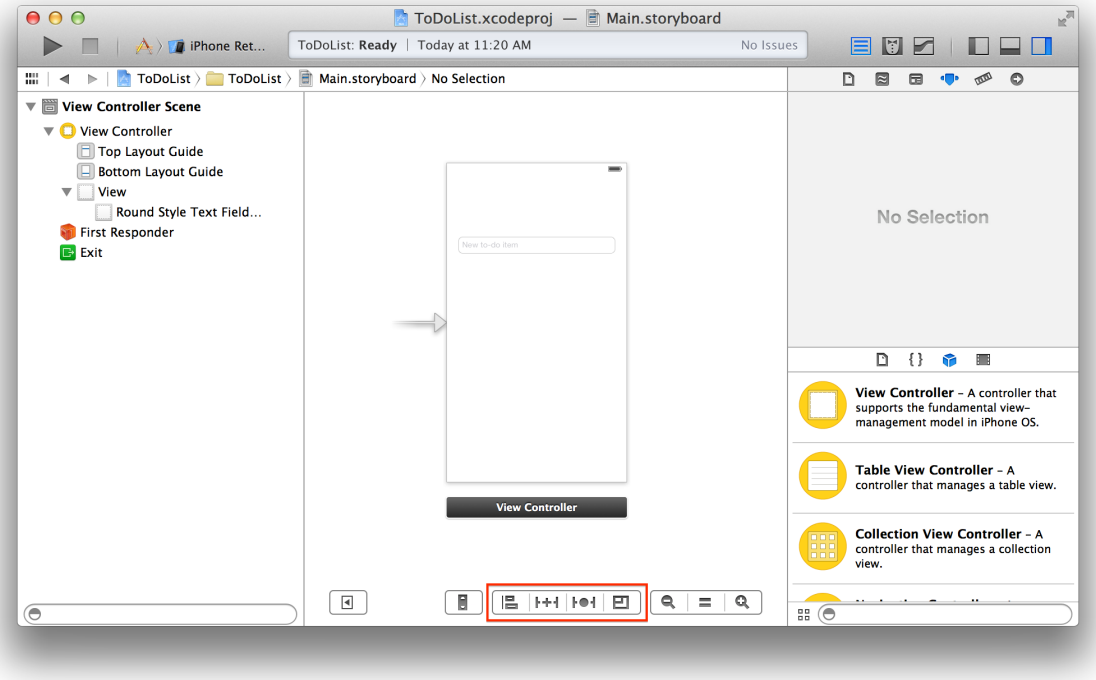
You began working with the Attributes inspector in the first tutorial. You'll continue using these inspectors throughout the rest of the tutorials to configure views and other objects in your storyboard. In particular, you'll use the Attributes inspector to configure your views, the Identity inspector to configure your view controllers, and the Connections inspector to create connections between your views and view controllers.

Use Auto Layout to Position Views

When you start positioning views in your storyboard, you need to consider a variety of situations. iOS apps run on a number of different devices, with various screen sizes, orientations, and languages. Instead of designing a static interface, you want it to be dynamic and to seamlessly respond to changes in screen size, device orientation, localization, and metrics.

To help you use views to create a versatile interface, Xcode offers a tool called Auto Layout. **Auto Layout** is a system for expressing relationships between views in your app's user interface. Auto Layout lets you define these relationships in terms of constraints on individual views or between sets of views.

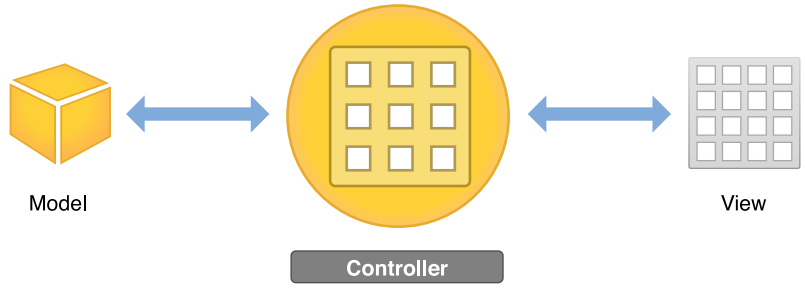
The Auto Layout menu, which resides in the bottom-right area of your canvas, has four segments. You use this menu to add various types of constraints to views on your canvas, resolve layout issues, and determine constraint resizing behavior.



You'll work briefly with Auto Layout in the second tutorial to add support for landscape mode to your ToDoList app.

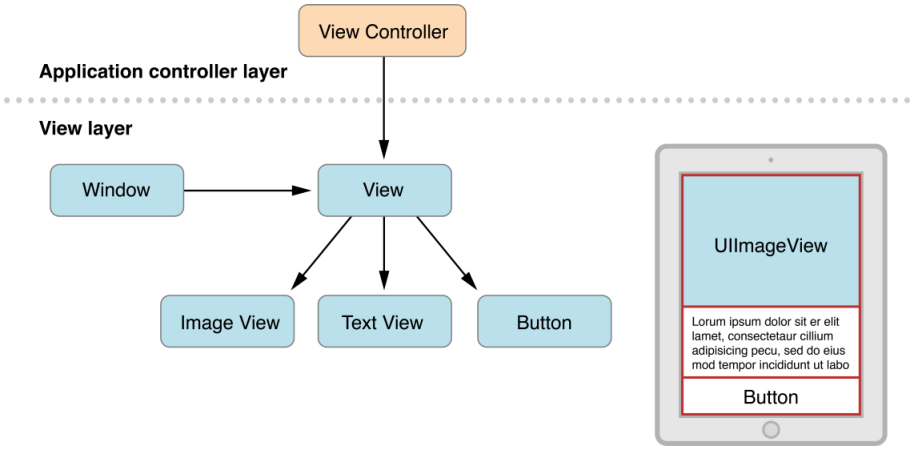
Defining the Interaction

After you lay out your user interface, you need to let users interact with it. This is where controllers come in. **Controllers** support your views by responding to user actions and populating the views with content. Controller objects are a conduit through which views learn about changes in the data model, and vice versa. Views are notified of changes in model data through the app's controllers, and controllers communicate user-initiated changes—for example, text entered in a text field—to model objects. Whether they're responding to user actions or defining navigation, controllers implement your app's behavior.



View Controllers

After you've built a basic view hierarchy, your next step is to control the visual elements and respond to user input. In an iOS app, you use a **view controller** (`UIViewController`) to manage a content view with its hierarchy of subviews.



A view controller isn't part of the view hierarchy and it's not an element in your interface. Instead, it manages the view objects in the hierarchy and provides them with behavior. Each content view hierarchy that you build in your storyboard needs a corresponding view controller, responsible for managing the interface elements and performing tasks in response to user interaction. This usually means writing a custom `UIViewController` subclass for each content view hierarchy. If your app has multiple content views, you use a different custom view controller class for each content view.

View controllers play many roles. They coordinate the flow of information between the app's data model and the views that display that data, manage the life cycle of their content views, and handle orientation changes when the device is rotated. But perhaps their most obvious role is to respond to user input.

You also use view controllers to implement transitions from one type of content to another. Because iOS apps have a limited amount of space in which to display content, view controllers provide the infrastructure needed to remove the views from one view controller and replace them with the views of another.

To define interaction in your app, make your view controller files communicate with the views in your storyboard. You do this by defining connections between the storyboard and source code files through actions and outlets.

Actions

An **action** is a piece of code that's linked to some kind of event that can occur in your app. When that event takes place, the code gets executed. You can define an action to accomplish anything from manipulating a piece of data to updating the user interface. You use actions to drive the flow of your app in response to user or system events.

You define an action by creating and implementing a method with an `IBAction` return type and a `sender` parameter.

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

The `sender` parameter points to the object that was responsible for triggering the action. The `IBAction` return type is a special keyword; it's like the `void` keyword, but it indicates that the method is an action that you can connect to from your storyboard in Interface Builder (which is why the keyword has the `IB` prefix). You'll learn more about how to link an `IBAction` action to an element in your storyboard in "Tutorial: Storyboards" (page 47).

Outlets

Outlets provide a way to reference objects from your interface—the objects you added to your storyboard—from source code files. You create an outlet by Control-dragging from a particular object in your storyboard to a view controller file. This creates a property for the object in your view controller file, which lets you access and manipulate that object from code at runtime. For example, in the second tutorial, you'll create an outlet for the text field in your ToDoList app to be able to access the text field's contents in code.

Outlets are defined as `IBOutlet` properties.

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

The `IBOutlet` keyword tells Xcode that you can connect to this property from Interface Builder. You'll learn more about how to connect an outlet from a storyboard to source code in "Tutorial: Storyboards" (page 47).

Controls

A **control** is a user interface object such as a button, slider, or switch that users manipulate to interact with content, provide input, navigate within an app, and perform other actions that you define. Controls provide a way for your code to receive messages from the user interface.

When a user interacts with a control, a control event is created. A **control event** represents various physical gestures that users can make on controls, such as lifting a finger from a control, dragging a finger onto a control, and touching down within a text field.

There are three general categories of event types:

- **Touch and drag events.** Touch and drag events occur when a user interacts with a control with a touch or drag. There are several available touch event stages. When a user initially touches a finger on a button, for example, the Touch Down Inside event is triggered; if the user drags out of the button, the respective drag events are triggered. Touch Up Inside is sent when the user lifts a finger off the button while still within the bounds of the button's edges. If the user has dragged a finger outside the button before lifting the finger, effectively canceling the touch, the Touch Up Outside event is triggered.
- **Editing events.** Editing events occur when a user edits a text field.
- **Value-changed events.** Value-changed events occur when a user manipulates a control, causing it to emit a series of different values.

As you define the interactions, know the action that's associated with every control in your app and then make that control's purpose obvious to users in your interface.

Navigation Controllers

If your app has more than one content view hierarchy, you need to be able to transition between them. For this, you'll use a specialized type of view controller: a navigation controller (`UINavigationController`). A **navigation controller** manages transitions backward and forward through a series of view controllers, such as when a user navigates through email accounts, inbox messages, and individual emails in the iOS Mail app.

The set of view controllers managed by a particular navigation controller is called its navigation stack. The **navigation stack** is a last-in, first-out collection of custom view controller objects. The first item added to the stack becomes the **root view controller** and is never popped off the stack. Other view controllers can be pushed on or popped off the navigation stack.

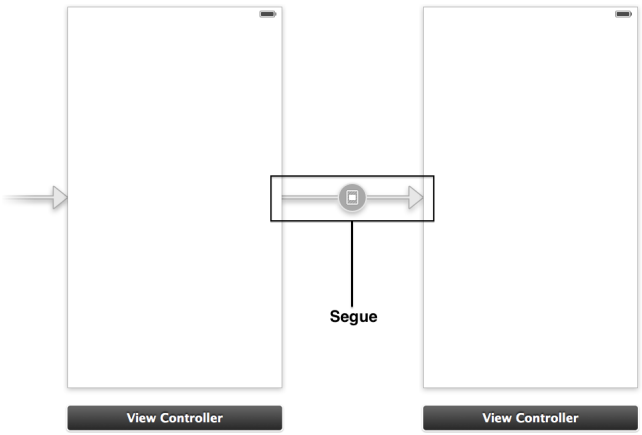
Although a navigation controller's primary job is to manage the presentation of your content view controllers, it's also responsible for presenting custom views of its own. Specifically, it presents a navigation bar—the view at the top of the screen that provides context about the user's place in the navigation hierarchy—which contains a back button and other buttons you can customize. Every view controller that's added to the navigation stack presents this navigation bar. You are responsible for configuring the navigation bar.

You generally don't have to do any work to pop a view controller off of the navigation stack; the back button provided by the navigation controller handles this for you. However, you do have to manually push a view controller onto the stack. You can do this using storyboards.

Use Storyboards to Define Navigation

So far, you've learned about using storyboards to create a single screen of content in your app. Now, you'll learn about using them to define the flow between multiple scenes in your app.

In the first tutorial, the storyboard you worked with had one scene. In most apps, a storyboard is composed of a sequence of scenes, each of which represents a view controller and its view hierarchy. Scenes are connected by **segues**, which represent a transition between two view controllers: the source and the destination.



There are several types of segues you can create:

- **Push.** A push segue adds the destination view controller to the navigation stack. Push segues may only be used when the source view controller is connected to a navigation controller.
- **Modal.** A modal segue is simply one view controller presenting another controller modally, requiring a user to perform some operation on the presented controller before returning to the main flow of the app. A modal view controller isn't added to a navigation stack; instead, it's generally considered to be a child of the presenting view controller. The presenting view controller is responsible for dismissing the modal view controller it created and presented.
- **Custom.** You can define your own custom transition by subclassing `UIStoryboardSegue`.
- **Unwind.** An unwind segue moves backward through one or more segues to return the user to an existing instance of a view controller. You use unwind segues to implement reverse navigation.

Instead of segues, scenes may also be connected by a **relationship**. For example, there's a relationship between the navigation controller and its root view controller. In this case, the relationship represents the containment of the root view controller by the navigation controller.

When you use a storyboard to plan the user interface for your app, it's important to make sure that one of the view controllers is marked as being the **initial view controller**. At runtime, this is the view controller whose content view will be displayed the first time the app is launched and from which you can transition to other view controllers' content views as necessary.

Now that you've learned the basics of working with views and view controllers in storyboards, it's time to incorporate this knowledge into your ToDoList app in the next tutorial.

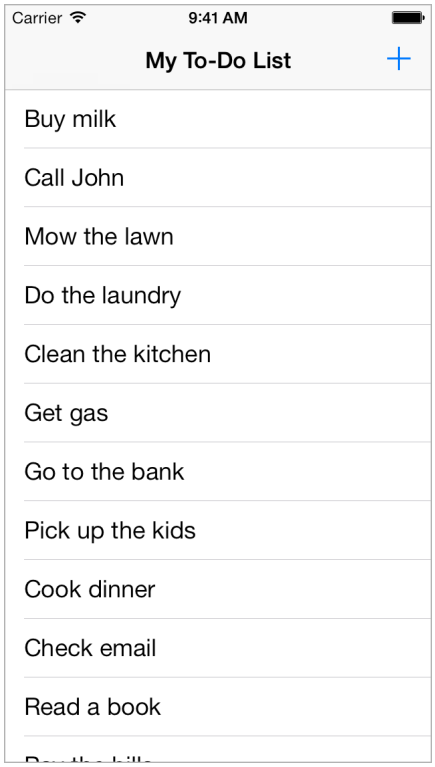
Tutorial: Storyboards

This tutorial builds on the project you created in the first tutorial ("Tutorial: Basics" (page 8)). You'll put to use what you learned about views, view controllers, actions, and navigation. Following the interface-first design process, you'll also create some of the key user interface flows for your ToDoList app and add behavior to the scene you've already created.

This tutorial teaches you how to:

- Adopt Auto Layout to add flexibility to your user interface
- Use storyboards to define app content and flow
- Manage multiple view controllers
- Add actions to elements in your user interface

After you complete all the steps in this tutorial, you'll have an app that looks something like this:

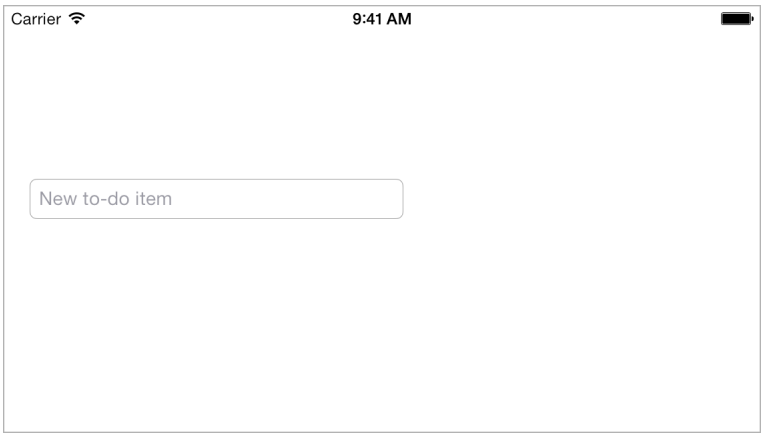


Adopt Auto Layout

The add-to-do-item scene is configured to work in portrait mode because that's how you created it. So what happens if a user rotates the device? Try it out by running your app in Simulator.

To rotate in iOS Simulator

1. Launch your app in iOS Simulator.
2. Choose Hardware > Rotate Left (or press Command–Left Arrow).



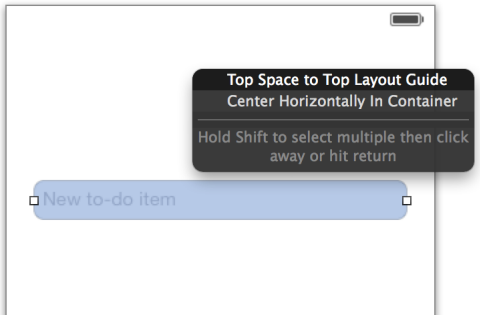
As you see, the text field doesn't look quite right. It stops about halfway across the screen. The text field should stretch all the way across the screen, as it does in portrait mode. Fortunately, Xcode has a powerful built-in layout engine called Auto Layout. With Auto Layout you describe your intent for the positioning of elements in a scene and then let the layout engine determine how best to implement that intent. You describe your intent using **constraints**—rules that explain where one element should be located relative to another, or what size it should be, or which of two elements should shrink first when something reduces the space available for each of them. For the add-to-do-item scene, two sets of constraints are needed—one to position the text field and the other to set its size.

Setting these constraints can easily be accomplished in Interface Builder.

To position the text field using Auto Layout

1. In the project navigator, select `Main.storyboard`.
2. In your storyboard, select the text field.
3. On the canvas, Control-drag from the text field toward the top of the scene, ending in the empty space around the text field. This space is the text field's superview.

A shortcut menu appears in the location where you release the drag.



4. Choose "Top Space to Top Layout Guide" from the shortcut menu.

   A spacing constraint is created between the top of the text field and the navigation bar.

   If a different shortcut menu appears, perhaps with a menu item such as "Leading Space to Container," this is because instead of dragging vertically to the top of the screen, you dragged in a different direction. Xcode uses the direction you drag in as a way to understand what kind of constraints you're trying to make, and it uses the start and end points of the drag to understand which objects are being related by the constraints. Go ahead and experiment with different dragging directions to see what constraints are available.

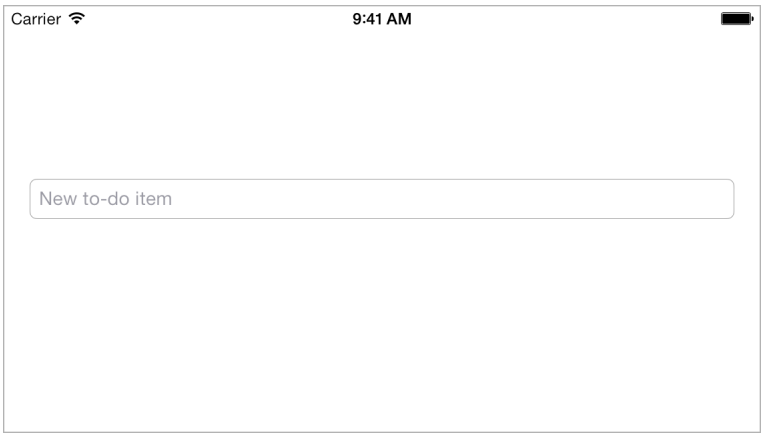
5. When you're done experimenting, Control-drag from the text field to the right, ending in the superview, to create a "Trailing Space to Container" constraint.

6. Control-drag from the text field to the left, ending in its superview, to create a "Leading Space to Container" constraint.

These constraints specify that the distance between the edges of the text field and its superview shouldn't change. This means that if the device orientation changes, the text field will automatically grow to satisfy these constraints.

Checkpoint: Run your app. If you rotate the device, the text field grows or shrinks to the appropriate size depending on the device's orientation.

If you don't get the behavior you expect, use the Xcode Auto Layout debugging features to help you. With the text field selected, choose Editor > Resolve Auto Layout Issues > "Reset to Suggested Constraints" to have Xcode set up the constraints described by the steps above. Or choose Editor > Resolve Auto Layout Issues > Clear Constraints to remove all constraints on the text view, and then try following the steps above again.



Although your add item scene doesn't do much yet, the basic user interface is there and functional. Considering layout from the start ensures that you have a solid foundation to build upon.

Creating a Second Scene

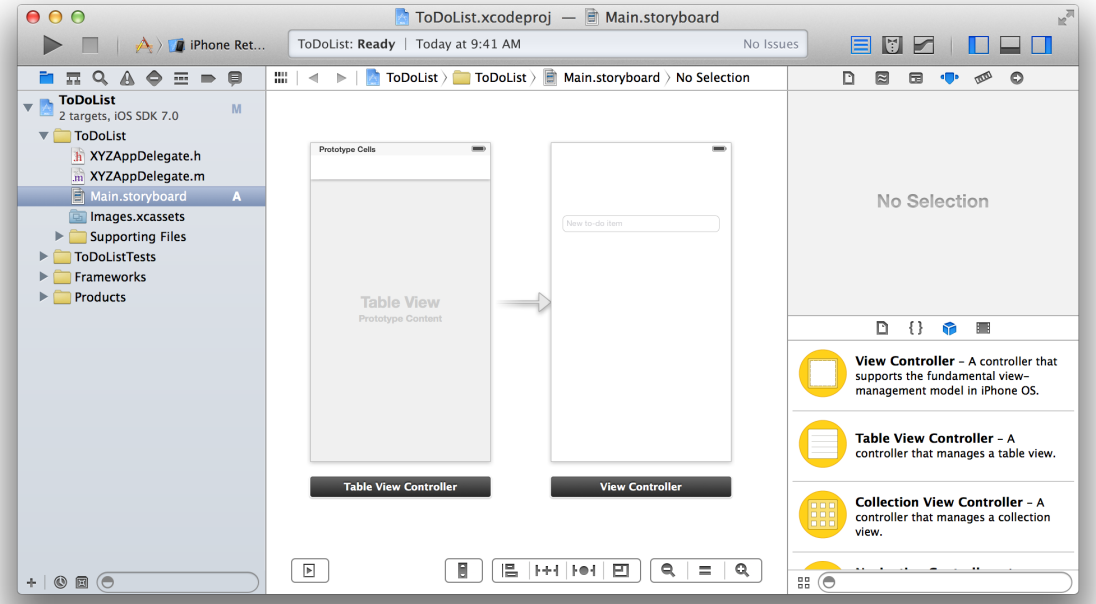
So far, you've been working with a single scene managed by a view controller that represents a page where you can add an item to your to-do list. Now it's time to create the scene that shows the entire to-do list. Fortunately, iOS comes with a powerful built-in class called a **table view** designed specifically to display a scrolling list of items.

To add a scene with a table view to your storyboard

1. In the project navigator, select `Main.storyboard`.
2. Open the Object library in the utility area. (To open the library with a menu command, choose View > Utilities > Show Object Library.)
3. Drag a Table View Controller object from the list and drop it on the canvas to the left of the add-to-do-item scene. If you need to, you can use the Zoom Out button in the lower right of the canvas to get enough space to drag it to.

   If you see a table view with content and nothing happens when you try to drag it to the canvas, you're probably dragging a table view rather than a table view controller. A table view is one of the things managed by a table view controller, but you want the whole package, so find the table view controller and drag it to the canvas.

You now have two scenes, one for displaying the list of to-do items and one for adding to-do items.

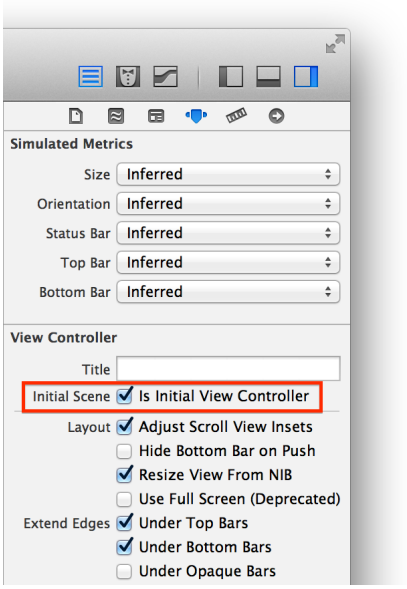


It makes sense to have the list be the first thing users see when they launch your app, so tell Xcode that's your intent by setting the table view controller as the first scene.

To set the table view controller as the initial scene

1. If necessary, open the outline view using the button in the lower left of the canvas.
2. In the outline view, select the newly added table view controller.
3. With the table view controller selected, open the Attributes inspector in the utility area.

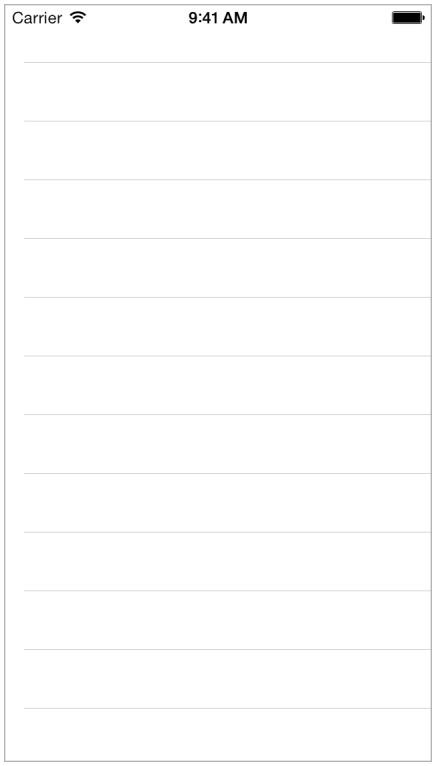
4. In the Attributes inspector, select the checkbox next to the Is Initial View Controller option.



Alternatively, you can drag the initial scene indicator from the `XYZAddToDoItemViewController` to the table view controller directly on the canvas.

The table view controller is set as the initial view controller in your storyboard, making it the first scene that loads on app launch.

Checkpoint: Run your app. Instead of the add-to-do-item scene with its text field, you should now see an empty table view—a screen with a number of horizontal dividers to separate it into rows, but with no content in each row.



Display Static Content in a Table View

Because you haven't learned about storing data yet, it's too early to create and store to-do items and display them in the table view. But you don't need real data to prototype your user interface. Xcode allows you to create static content in a table view in Interface Builder. This makes it a lot easier to see how your user interface will behave, and it's a valuable way to try out different ideas.

To create a static cell in your table view

1. In the outline view for your interface, select Table View under Table View Controller.
2. With the table view selected, open the Attributes inspector in the utility area.
3. In the Attributes inspector, choose Static Cells from the pop-up menu next to the Content option.

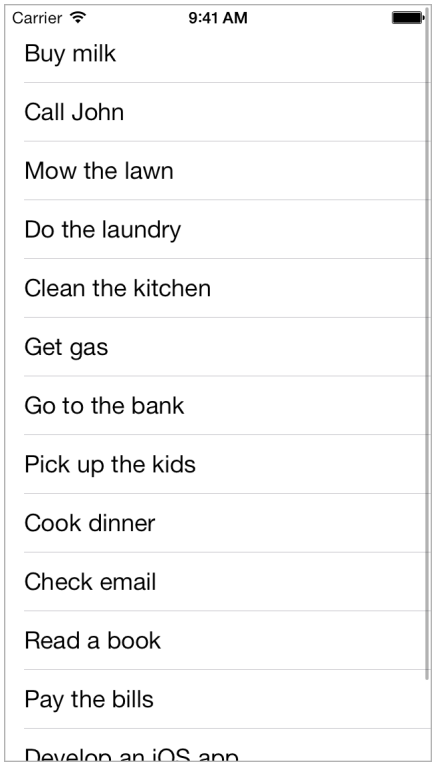
   Three empty table view cells appear in your table view.
4. In the outline view or on the canvas, select the top cell.
5. In the Attributes inspector, choose Basic from the pop-up menu next to the Style option.

The Basic style includes a label, so Xcode creates a label with the text "Title" in the table cell.

6. In the outline view or on the canvas, select the label.
7. In the Attributes inspector, change the text of the label from "Title" to "Mow the Lawn." For the change to take effect, press Enter or click outside the utility area.

   Alternatively, you can edit a label by double-clicking it and editing the text directly.
8. Repeat steps 4–7 for the other cells, giving them text for other likely to-do items.
9. Create enough cells so that the items more than fill the screen. You can create new cells by copying and pasting them or by holding down the Option key when dragging a cell.

Checkpoint: Run your app. You should now see a table view with the preconfigured cells you added in Interface Builder. See how the new table view feels when you scroll it. Try rotating the simulated device—notice how the table view is already configured to lay out its contents properly. You get a lot of behavior for free by using a table view.



When you're done, it's time to provide a way to navigate from this table view, with its list of to-do items, to the first scene you created, where a user can create a new to-do item.

Add a Segue to Navigate Forward

You have two view controllers configured in the storyboard, but there's no connection between them. Transitions between scenes are called **segues**.

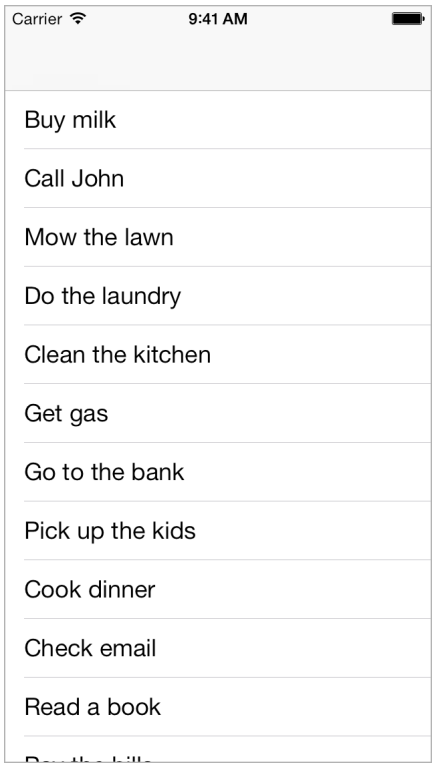
Before creating a segue, you need to configure your scenes. First, you'll wrap your `XYZToDoListViewController` in a navigation controller. Recall from "Defining the Interaction" (page 42) that navigation controllers provide a navigation bar and keep track of the navigation stack. You'll add a button to this navigation bar to transition to the `XYZAddToDoItemViewController` scene.

To add a navigation controller to your table view controller

1. In the outline view, select Table View Controller.
2. With the view controller selected, choose Editor > Embed In > Navigation Controller.

Xcode adds a new navigation controller to your storyboard, sets the initial scene to it, and creates a relationship between the new navigation controller and your existing table view controller. On the canvas, if you select the icon connecting the two scenes, you'll see that it's the root view controller relationship. This means that the view for the content displayed below the navigation bar will be your table view. The initial scene is set to the navigation controller because the navigation controller holds all of the content that you'll display in your app—it's the container for both the to-do list and the add-to-do-item scenes.

Checkpoint: Run your app. Above your table view you should now see extra space. This is the navigation bar provided by the navigation controller.



Now, you'll add a title (to the to-do list) and a button (to add additional to-do items) to the navigation bar.

To configure the navigation bar

1. In the outline view or on the canvas, select Navigation Item under Table View Controller.

   Navigation bars get their title from the view controller that the navigation controller currently displays—they don't themselves have a title. You set the title using the navigation item of your to-do list (the table view controller) rather than setting it directly on the navigation bar.

2. In the Attributes inspector, type `My To-Do List` in the Title field.

3. If necessary, open the Object library.

4. Drag a bar button item from the list to the far right of the navigation bar in the table view controller.

   A button containing the text "Item" appears where you dragged the bar button item.

5. In the outline view or on the canvas, select the bar button item.

6. In the Attributes inspector, find the Identifier option in the Bar Button Item section. Choose Add from the Identifier pop-up menu.

   The button changes to an Add button (+).

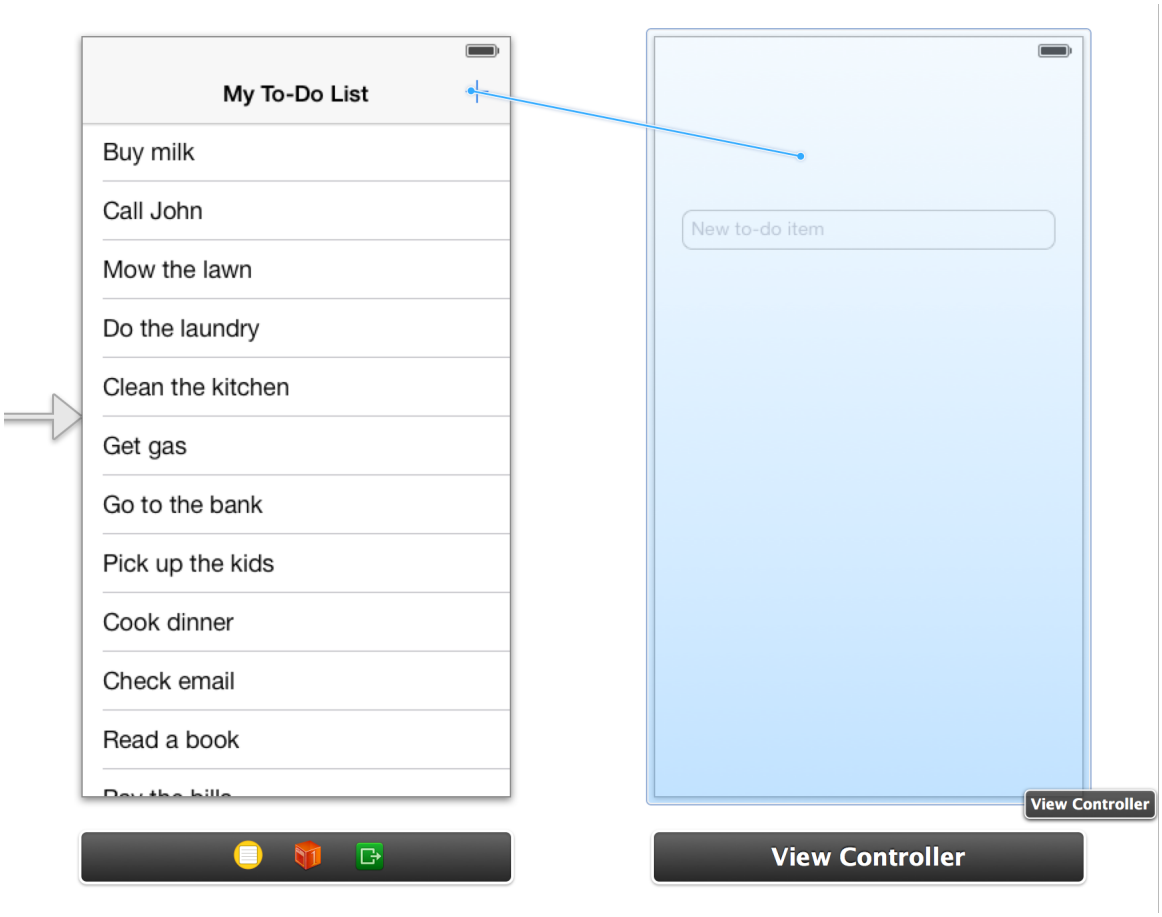
Checkpoint: Run your app. The navigation bar should now have a title and display an Add button. The button doesn't do anything yet. You'll fix that next.

You want the Add button to bring up the add-to-do-item scene. The scene is already configured—it was the first scene you created—but it's not connected to the other scenes. Xcode makes it easy to configure the Add button to bring up another scene when tapped.

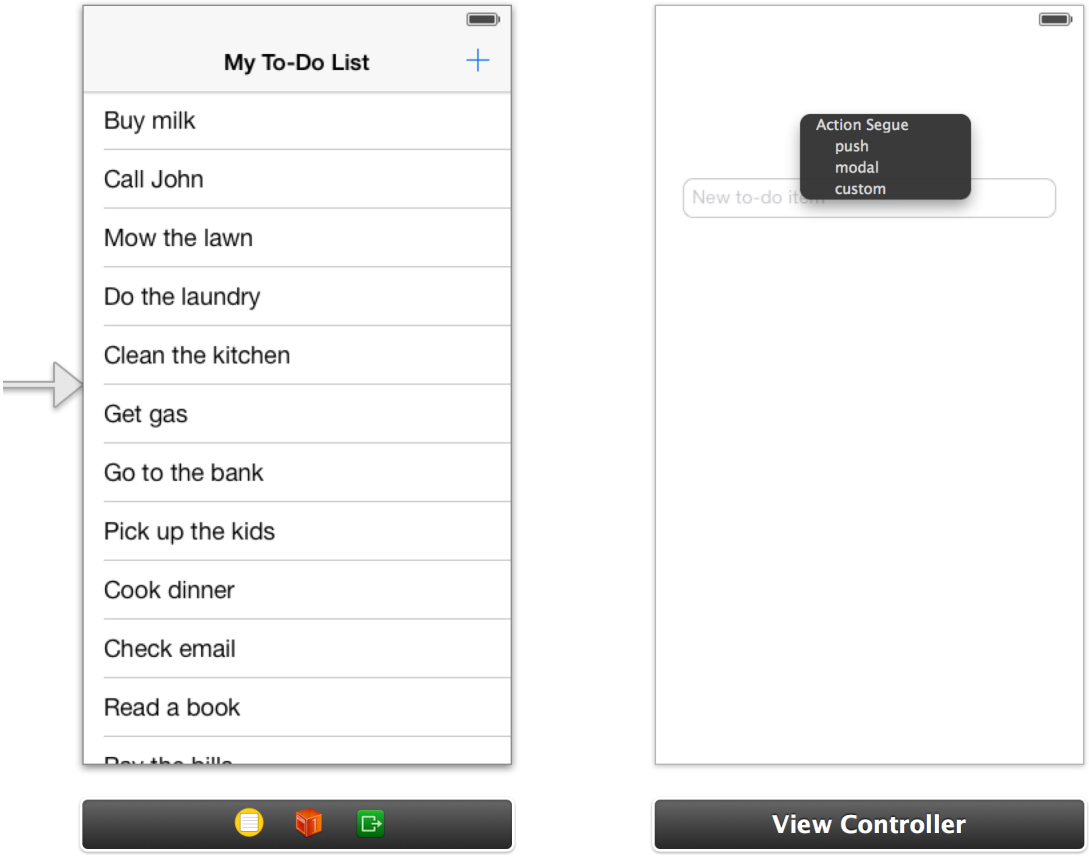
To configure the Add button

1. On the canvas, select the Add button.

2. Control-drag from the button to the add-to-do-item view controller.



A shortcut menu titled Action Segue appears in the location where the drag ended.

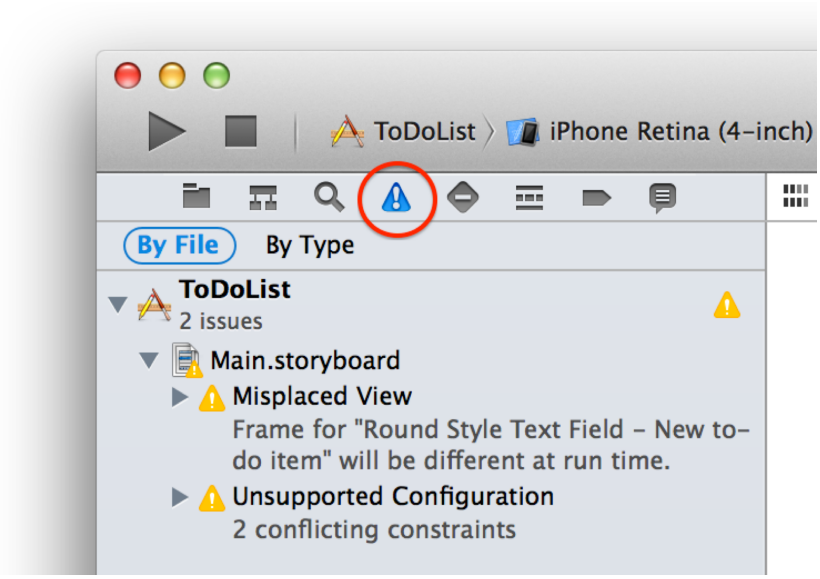


This is how Xcode allows you to choose what type of segue should be used to transition from the to-do list to the add-to-do-item view controller when the user taps the Add button.

3. Choose “push” from the shortcut menu.

Xcode sets up the segue and configures the add-to-do-item view controller to be displayed in a navigation controller—you’ll see the navigation bar in Interface Builder.

At this point, you might notice a couple of warnings in your project. Go ahead and open the Issue navigator to see what's wrong.



Because you added the add-to-do-item scene to the navigation stack, it now displays a navigation bar. This bar caused the frame of your text field to move down, which means that the Auto Layout constraints you specified earlier are no longer satisfied. Fortunately, this is easy to fix.

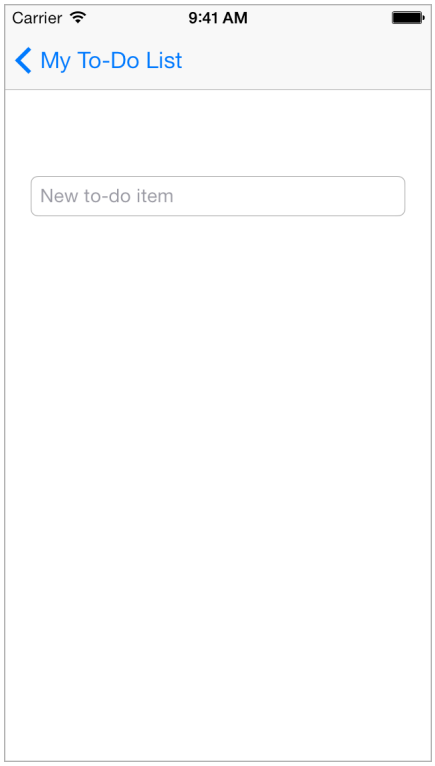
To update the Auto Layout constraints

1. In the outline view or on the canvas, select the text field.
2. On the canvas, open the Resolve Auto Layout Issues pop-up menu , and choose Update Constraints.

   Alternatively, you can choose Editor > Resolve Auto Layout Issues > Update Constraints.

   The constraints are updated and the Xcode warnings disappear.

Checkpoint: Run your app. You can click the Add button and navigate to the add-to-do-item view controller from the table view. Because you're using a navigation controller with a push segue, the backward navigation is handled for you. This means you can click the back button to get back to the table view.



The push navigation is working just as it's supposed to—but it's not quite what you want when adding items. Push navigation is designed for a drill-down interface, where you're providing more information about whatever the user selected. Adding an item, on the other hand, is a modal operation—the user performs some action that's complete and self-contained, and then returns from that scene to the main navigation. The appropriate method of presentation for this type of scene is a modal segue.

To change the segue style

1. In the outline view or on the canvas, select the segue from the table view controller to the add-to-do-item view controller.
2. In the Attributes inspector, choose Modal from the pop-up menu next to the Style option.

Because a modal view controller doesn't get added to the navigation stack, it doesn't get a navigation bar from the table view controller's navigation controller. However, you want to keep the navigation bar to provide the user with visual continuity. To give the add-to-do-item view controller a navigation bar when presented modally, embed it in its own navigation controller.

To add a navigation controller to the add-to-do-item view controller

1. In the outline view, select View Controller.
2. With the view controller selected, choose Editor > Embed In > Navigation Controller.

As before, Xcode adds a navigation controller and shows the navigation bar at the top of the view controller. Next, configure this bar to add a title to this scene as well as two buttons, Cancel and Done. Later, you'll link these buttons to actions.

To configure the navigation bar in the add-to-do-item view controller

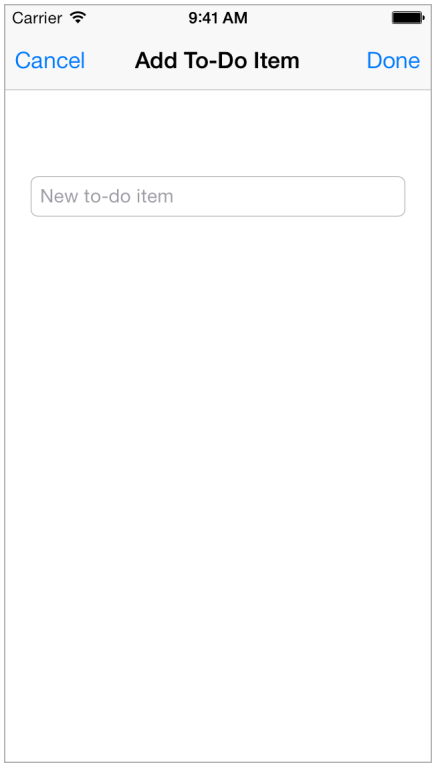
1. In the outline view or on the canvas, select Navigation Item under View Controller. If necessary, open the Attributes inspector .
2. In the Attributes inspector, type `Add To-Do Item` in the Title field.

   Xcode changes the description of the view controller from "View Controller" to "View Controller – Add To-Do Item" to make it easier for you to identify the scene. The description appears in the outline view.
3. Drag a bar button item from the Object library to the far right of the navigation bar in the add-to-do-item view controller.
4. In the Attributes inspector, choose Done from the pop-up menu next to the Identifier option.

   The button text changes to "Done."
5. Drag another bar button item from the Object library to the far left of the navigation bar in the add-to-do-item view controller.
6. In the Attributes inspector, choose Cancel from the pop-up menu next to the Identifier option.

   The button text changes to "Cancel."

Checkpoint: Run your app. Click the Add button. You still see the add item scene, but there's no longer a button to navigate back to the to-do list—instead, you see the two buttons you added, Done and Cancel. Those buttons aren't linked to any actions yet, so you can click them, but they don't do anything. Configuring the buttons to complete or cancel editing of the new to-do item—and bring the user back to the to-do list—is the next task.



Create Custom View Controllers

You've accomplished all of this configuration without needing to write any code. Configuring the completion of the add-to-do-item view controller requires some code, though, and you need a place to put it. Right now Xcode has configured both the add-to-do-item view controller and the table view controller as generic view controllers. To have a place for your custom code, you need to create subclasses for each of these view controllers and then configure the interface to use those subclasses.

First, you'll address the add-to-do-item view controller scene. The custom view controller class will be called `XYZAddToDoItemViewController`, because this view controller will control the scene that adds a new item to your to-do list.

To create a subclass of UIViewController

1. Choose File > New > File (or press Command-N).

2. On the left of the dialog that appears, select the Cocoa Touch template under iOS.
3. Select Objective-C Class, and click Next.
4. In the Class field, type `AddToDoItem` after the `XYZ` prefix.
5. Choose `UIViewController` in the "Subclass of" pop-up menu.

   The class title changes to "XYZAddToDoItemViewController." Xcode helps you by making it clear from the naming that you're creating a custom view controller. That's great, so leave the new name as is.
6. Make sure the "Targeted for iPad" and "With XIB for user interface" options are unselected.
7. Click Next.
8. The save location will default to your project directory. Leave that as is.
9. The Group option will default to your app name, ToDoList. Leave that as is.
10. The Targets section will default to having your app selected and the tests for your app unselected. That's perfect, so leave that as is.
11. Click Create.

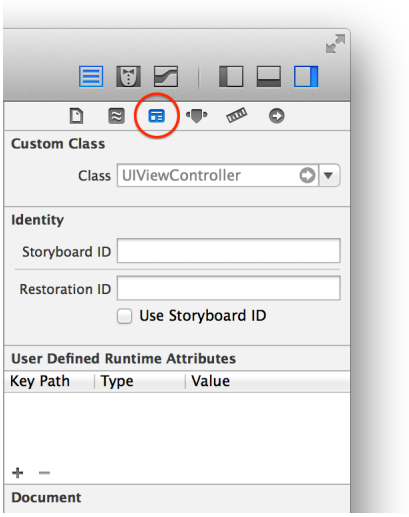
Now that you've created a custom view controller subclass, you need to tell your storyboard to use your custom class instead of the generic view controller. The storyboard file is a configuration of objects that's used at runtime by your app. The app machinery is smart enough to substitute your custom view controller for the generic view controller the storyboard starts with, but you need to tell the storyboard that that's what you want.

To identify your class as the view controller for a scene

1. In the project navigator, select `Main.storyboard`.
2. If necessary, open the outline view  .
3. In the outline view, select the "View Controller – Add To-Do Item" view controller.

   Click the disclosure triangle next to the "View Controller – Add To-Do Item" scene to show the objects in your scene. The first one should be the view controller. Click it to select it. Notice that the scene row has a different icon from the view controller row.
4. With the view controller selected, open the Identity inspector  in the utility area.

The **Identity inspector** appears at the top of the utility area when you click the third button from the left. It lets you edit properties of an object in your storyboard related to that object's identity, such as what class it is.



5. In the Identity inspector, open the pop-up menu next to the Class option.

   You'll see a list of all the view controller classes Xcode knows about. The last one in the list should be your custom view controller, `XYZAddToDoItemViewController`. Choose it to tell Xcode to use your view controller for this scene.

At runtime your storyboard will create an instance of `XYZAddToDoItemViewController`—your custom view controller subclass—instead of the generic `UIViewController`. Notice that Xcode changed the description of your add-to-do-item view controller scene from "View Controller – Add To-Do Item" to "Add To Do Item View Controller – Add To-Do Item." Xcode knows that you're now using a custom view controller for this scene, and it interprets the name of the custom class to make it easier to understand what's going on in the storyboard.

Now, do the same thing for the table view controller.

To create a subclass of UITableViewController

1. Choose File > New > File (or press Command-N).
2. On the left, select Cocoa Touch under iOS, then select Objective-C Class. If you haven't created any classes since the last steps in the tutorial, it's probably already selected for you.
3. Click Next.
4. In the Class field, type `ToDoList`. Notice that Xcode put the insertion point in between `XYZ`, your class prefix, and `ViewController`, the type of thing you're creating.
5. Choose `UITableViewController` in the "Subclass of" pop-up menu.
6. Make sure the "Targeted for iPad" and "With XIB for user interface" options are unselected.

7. Click Next.

   The save location will default to your project directory. Leave that as is.

8. The Group option will default to your app name, ToDoList. Leave that as is.

9. The Targets section will default to having your app selected and the tests for your app unselected. That's perfect, so leave that as is.

10. Click Create.

Once again, you need to make sure to configure your custom table view controller, `XYZToDoListViewController`, in your storyboard.

To configure your storyboard

1. In the project navigator, select `Main.storyboard`.
2. If necessary, open the outline view.
3. In the outline view, select the table view controller and open the Identity inspector in the utility area.
4. In the Identity inspector, choose `XYZToDoListViewController` from the pop-up menu next to the Class option.

Now, you're ready to add custom code to your view controllers.

Unwind a Segue to Navigate Back

In addition to push and modal segues, Xcode provides an **unwind segue**. This segue allows users to go from a given scene back to a previous scene, and it provides a place for you to add your own code that gets executed when users navigate between those scenes. You can use an unwind segue to navigate back from `XYZAddToDoItemViewController` to `XYZToDoListViewController`.

An unwind segue is created by adding an action method to the destination view controller (the view controller you want to unwind to). A method that can be unwound to must return an action (`IBAction`) and take in a storyboard segue (`UIStoryboardSegue`) as a parameter. Because you want to unwind back to `XYZToDoListViewController`, you need to add an action method with this format to the `XYZToDoListViewController` implementation.

To unwind back to XYZToDoListViewController

1. In the project navigator, open `XYZToDoListViewController.m`.
2. Add the following code below the `@implementation` line:

```
- (IBAction)unwindToList:(UIStoryboardSegue *)segue
{


}
```

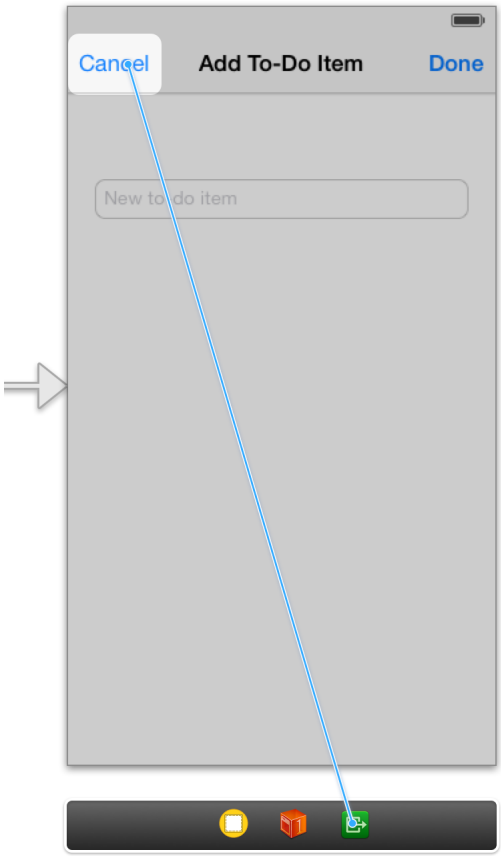
You can name the unwind action anything you want. Calling it `unwindToList:` makes it clear where the unwind will take you. For future projects, adopt a similar naming convention, one where the name of the action makes it clear where the unwind will go.

For now, leave this method implementation empty. Later, you'll use this method to retrieve data from the `XYZAddToDoItemViewController` to add an item to your to-do list.

To create the unwind segue, link the Cancel and Done buttons to the `unwindToList:` action through the Exit icon in the dock of the source view controller, `XYZAddToDoItemViewController`.

To link buttons to the unwindToList: action

1. In the project navigator, select `Main.storyboard`.
2. On the canvas, Control-drag from the Cancel button to the Exit item in the add-to-do-item scene dock.



If you don't see the Exit item in the scene dock but instead see the description of the scene, click the Zoom In button on the canvas until you see it.

A menu appears in the location where the drag ended.

3. Choose `unwindToList:` from the shortcut menu.

   This is the action you just added to the `XYZToDoListViewController.m` file. This means that when the Cancel button is tapped, the segue will unwind and this method will be called.

4. On the canvas, Control-drag from the Done button to the Exit item in the `XYZAddToDoItemViewController` scene dock.

5. Choose `unwindToList:` from the shortcut menu.

Notice that you used the same action for both the Cancel and the Done buttons. In the next tutorial, you'll distinguish between the two different cases when you write the code to handle the unwind segue.

Checkpoint: Now, run your app. At launch, you see a table view—but there's no data in it. You can click the Add button and navigate to `XYZAddToDoItemViewController` from `XYZToDoListViewController`. You can click the Cancel and Done buttons to navigate back to the table view.

So why doesn't your data show up? Table views have two ways of getting data—statically or dynamically. When a table view's controller implements the required `UITableViewDataSource` methods, the table view asks its view controller for data to display, regardless of whether static data has been configured in Interface Builder. If you look at `XYZToDoListViewController.m`, you'll notice that it implements three methods—`numberOfSectionsInTableView:`, `tableView:numberOfRowsInSection:`, and `tableView:cellForRowAtIndexPath:`. You can get your static data to display again by commenting out the implementations of these methods. Go ahead and try that out if you like.

Recap

At this point, you've finished developing the interface for your app. You have two scenes—one for adding items to your to-do list and one for viewing the list—and you can navigate between them. Next, you'll implement the ability to have users add a new to-do item and have it appear in the list. The next module covers working with data to implement this behavior.