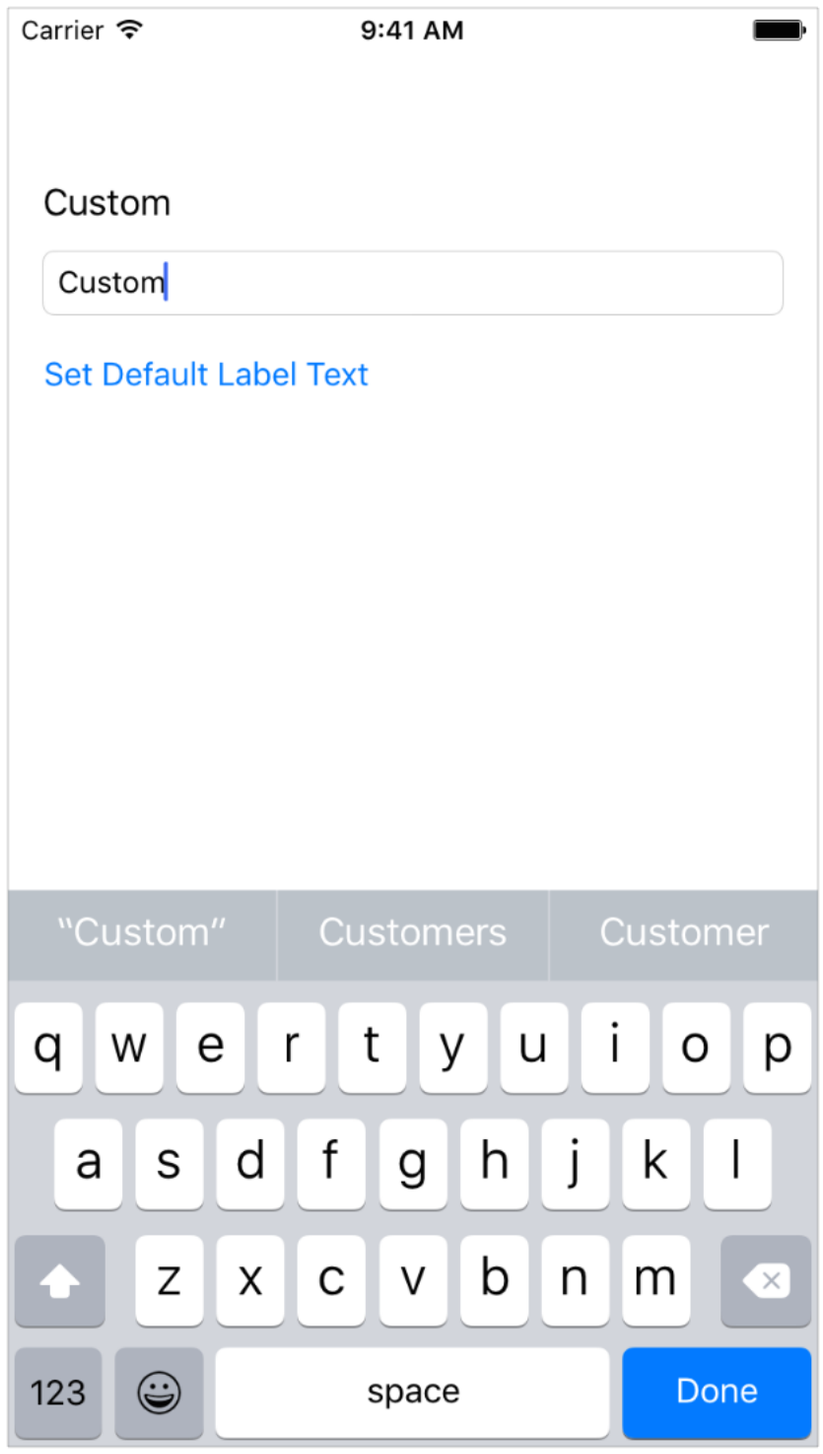
Connect the UI to Code

In this lesson, you'll connect the basic UI of the FoodTracker app to code and define some actions a user can perform on that UI. When you're finished, your app will look something like this:



Learning Objectives

At the end of the lesson, you'll be able to:

- Explain the relationship between a scene in a storyboard and the underlying view controller
- Create outlet and action connections between UI elements in a storyboard and source code
- Process user input from a text field and display the result in the UI
- Make a class conform to a protocol
- Understand the delegation pattern
- Follow the target-action pattern when designing app architecture

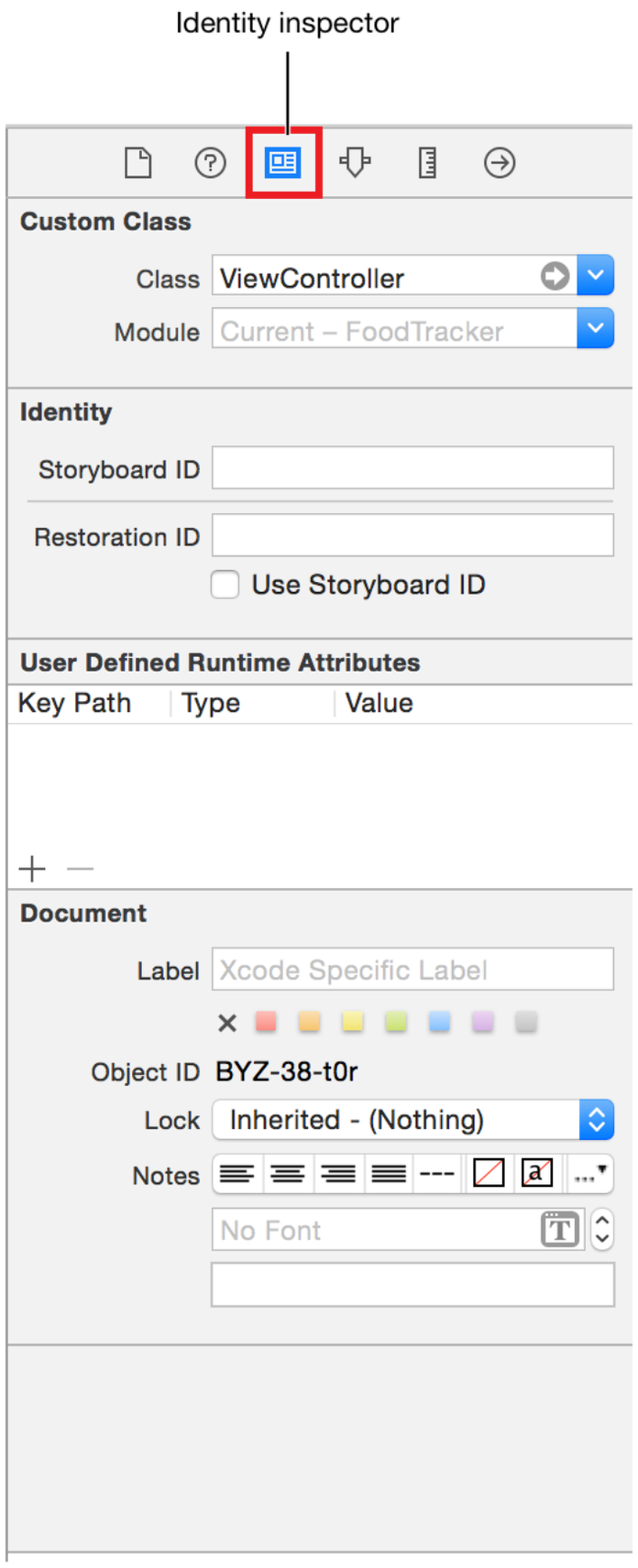
Connect the UI to Source Code

Elements in a storyboard are linked to source code. It's important to understand the relationship that a storyboard has to the code you write.

In a storyboard, a scene represents one screen of content and typically one view controller. View controllers implement your app's behavior. A view controller manages a single content view with its hierarchy of subviews. View controllers coordinate the flow of information between the app's data model, which encapsulates the app's data, and the views that display that data, manage the life cycle of their content views, handle orientation changes when the device is rotated, define the navigation within your app, and implement the behavior to respond to user input. All view controller objects in iOS are of type `UIViewController` or one of its subclasses.

You define the behavior of your view controllers in code by creating and implementing custom view controller subclasses. You can then create a connection between those classes and scenes in your storyboard to get the behavior you defined in code and the UI you defined in your storyboard.

Xcode already created one such class that you looked at earlier, `ViewController.swift`, and connected it to the scene you're working on in your storyboard right now. In the future, as you add more scenes, you'll make this connection yourself in the Identity inspector. The Identity inspector lets you edit properties of an object in your storyboard related to that object's identity, such as what class the object belongs to.



At runtime, your storyboard will create an instance of `ViewController`, your custom view controller subclass. The screen that you see in the app will show the UI defined in this scene in your storyboard and any behavior defined in `ViewController.swift`.

Although the scene is connected to `ViewController.swift`, that's not the only connection that needs to be made. To define interaction in your app, your view controller source code needs to be able to communicate with the views in your storyboard. You do this by defining additional connections—called outlets and actions—between the views in the storyboard and the view controller source code files.

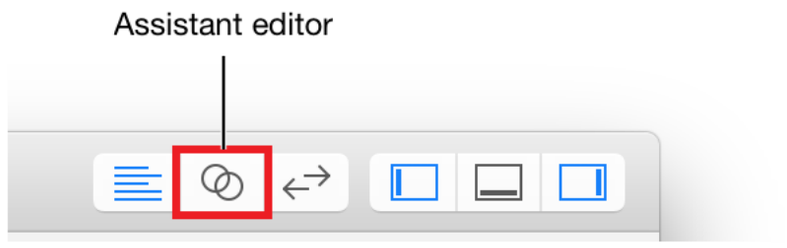
Create Outlets for UI Elements

Outlets provide a way to reference interface objects—the objects you added to your storyboard—from source code files. To create an outlet, Control-drag from a particular object in your storyboard to a view controller file. This operation creates a property for the object in your view controller file, which lets you access and manipulate that object from code at runtime.

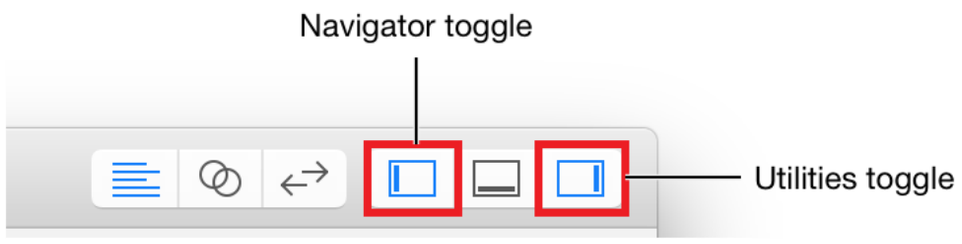
You'll need to create outlets for the text field and label in your UI to be able to reference them.

To connect the text field to the ViewController.swift code

1. Open your storyboard, `Main.storyboard`.
2. Click the Assistant button in the Xcode toolbar near the top right corner of Xcode to open the assistant editor.

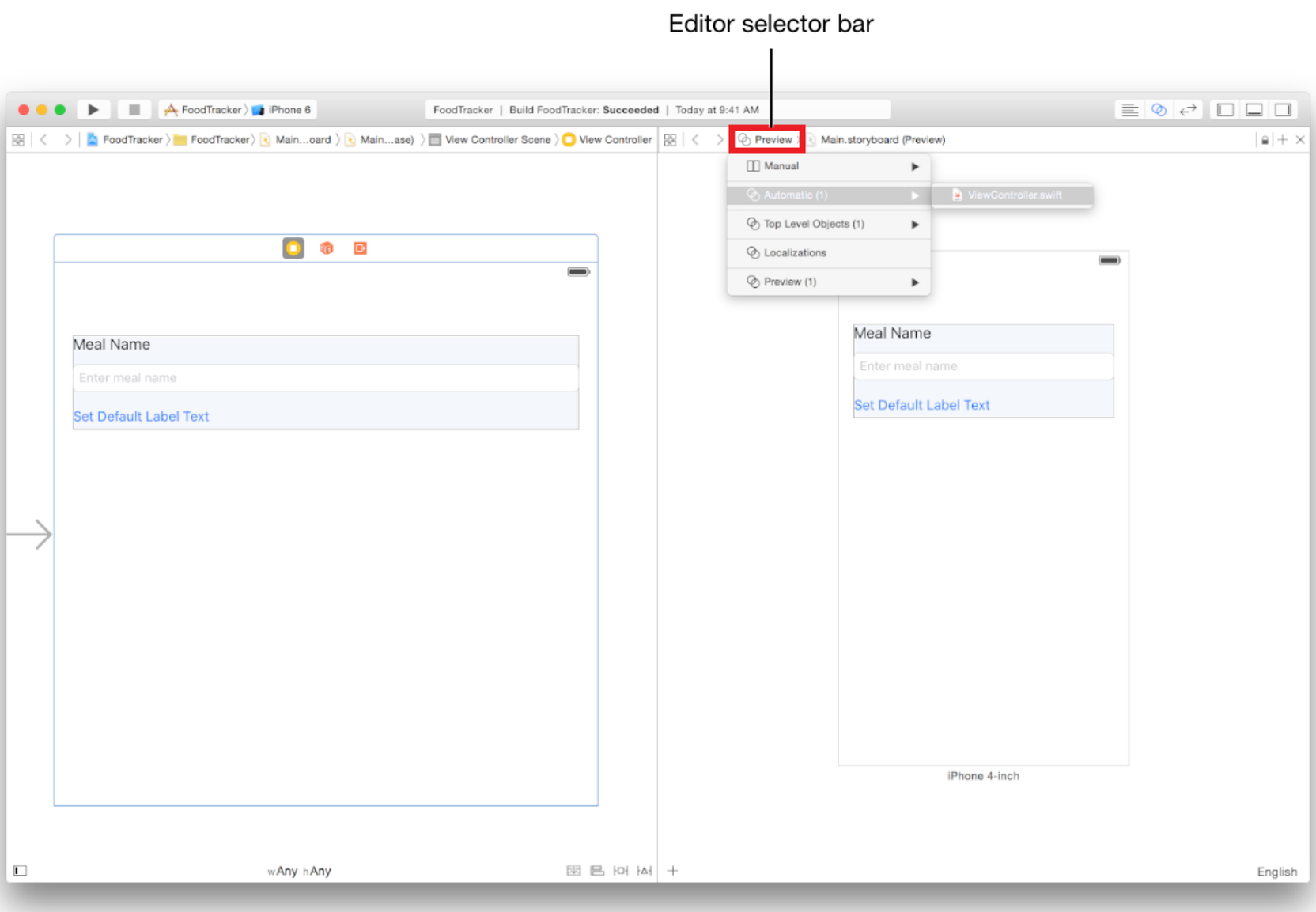


3. If you want more space to work, collapse the project navigator and utility area by clicking the Navigator and Utilities buttons in the Xcode toolbar.



   You can also collapse the outline view.

4. In the editor selector bar, which appears at the top of the assistant editor, change the assistant editor from Preview to Automatic > `ViewController.swift`.



   `ViewController.swift` displays in the editor on the right.

5. In `ViewController.swift`, find the `class` line, which should look like this:

   ```
   class ViewController: UIViewController {
   ```

6. Below the `class` line, add the following:

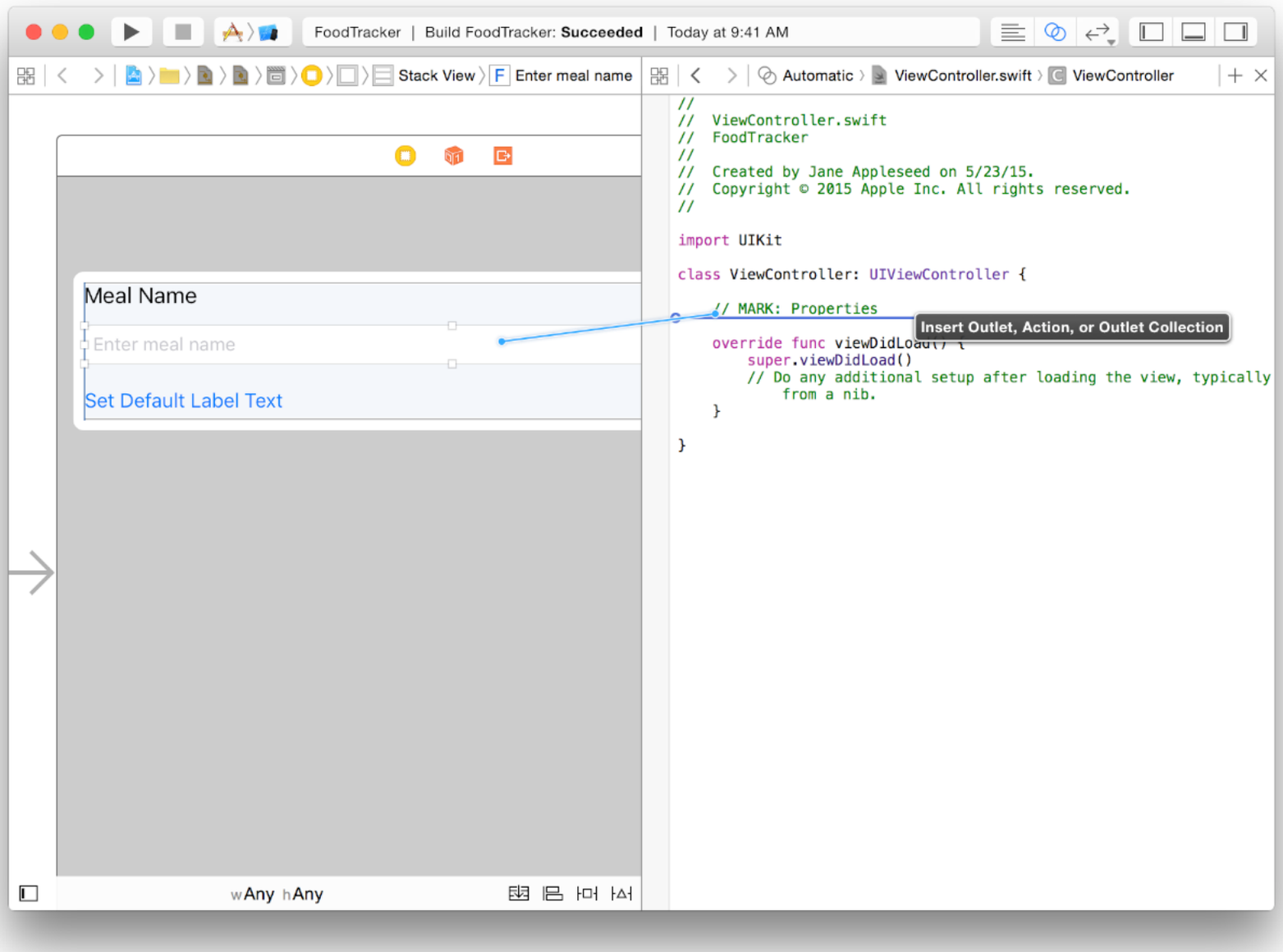
   ```
   // MARK: Properties
   ```

   You just added a comment to your source code. Recall that a comment is a piece of text in a source code file that doesn't get compiled as part of the program but provides context or useful information about individual pieces of code.

   A comment that begins with the characters `// MARK:` is a special type of comment that's used to organize your code and to help you (and anybody else who reads your code) navigate through it. You'll see this in action later. Specifically, the comment you added indicates that this is the section of your

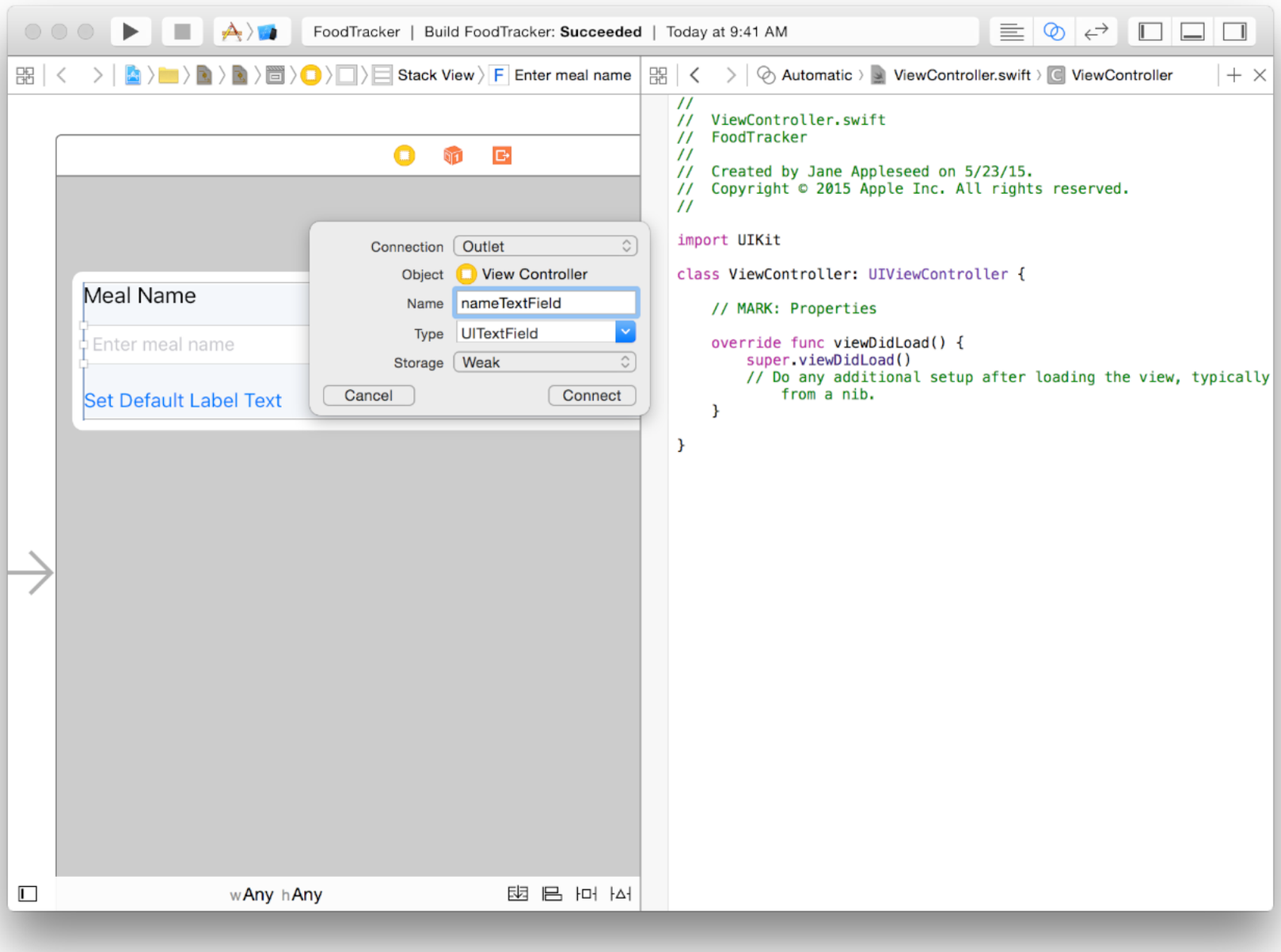
code that lists properties.

7. In your storyboard, select the text field.
8. Control-drag from the text field on your canvas to the code display in the editor on the right, stopping the drag at the line below the comment you just added in `ViewController.swift`.



9. In the dialog that appears, for Name, type `nameTextField`.

   Leave the rest of the options as they are. Your dialog should look like this:



10. Click Connect.

Xcode adds the necessary code to `ViewController.swift` to store a pointer to the text field and configures the storyboard to set up that connection.

```
@IBOutlet weak var nameTextField: UITextField!
```

Take a minute to understand what's happening in this line of code.

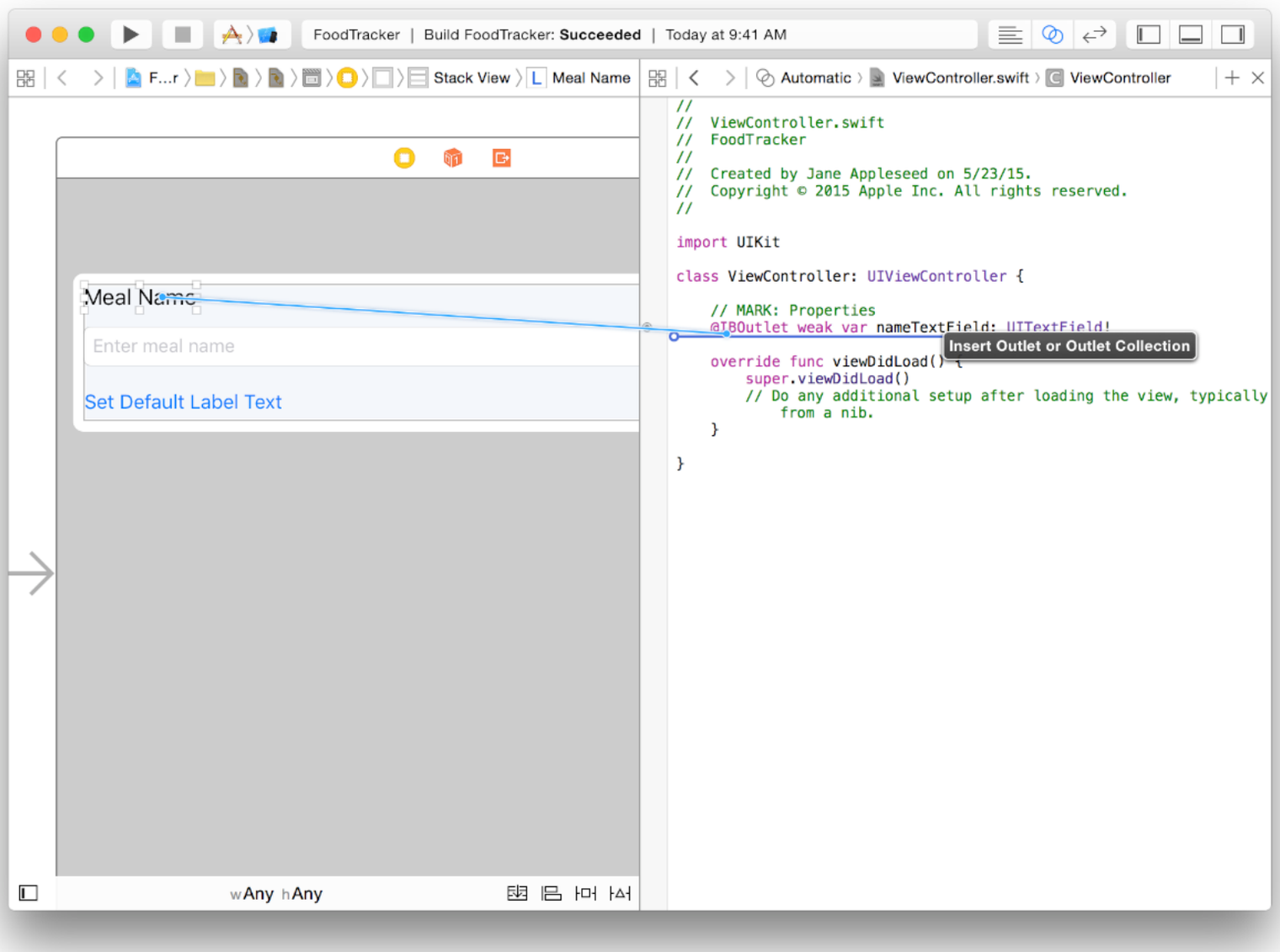
The `IBOutlet` attribute tells Xcode that you can connect to the `nameTextField` property from Interface Builder (which is why the attribute has the `IB` prefix). The `weak` keyword means that it's possible for that property to have no value (be `nil`) at some point in its life. The rest of the declaration declares a variable of type `UITextField` named `nameTextField`.

Note the exclamation point at the end of the declaration. You may remember seeing these at the end of some types in the `AppDelegate.swift` file. This exclamation point indicates that the type is an implicitly unwrapped optional, which is an optional type that will always have a value after the value is first set.

Now, connect the label to your code in the same way you connected the text field.

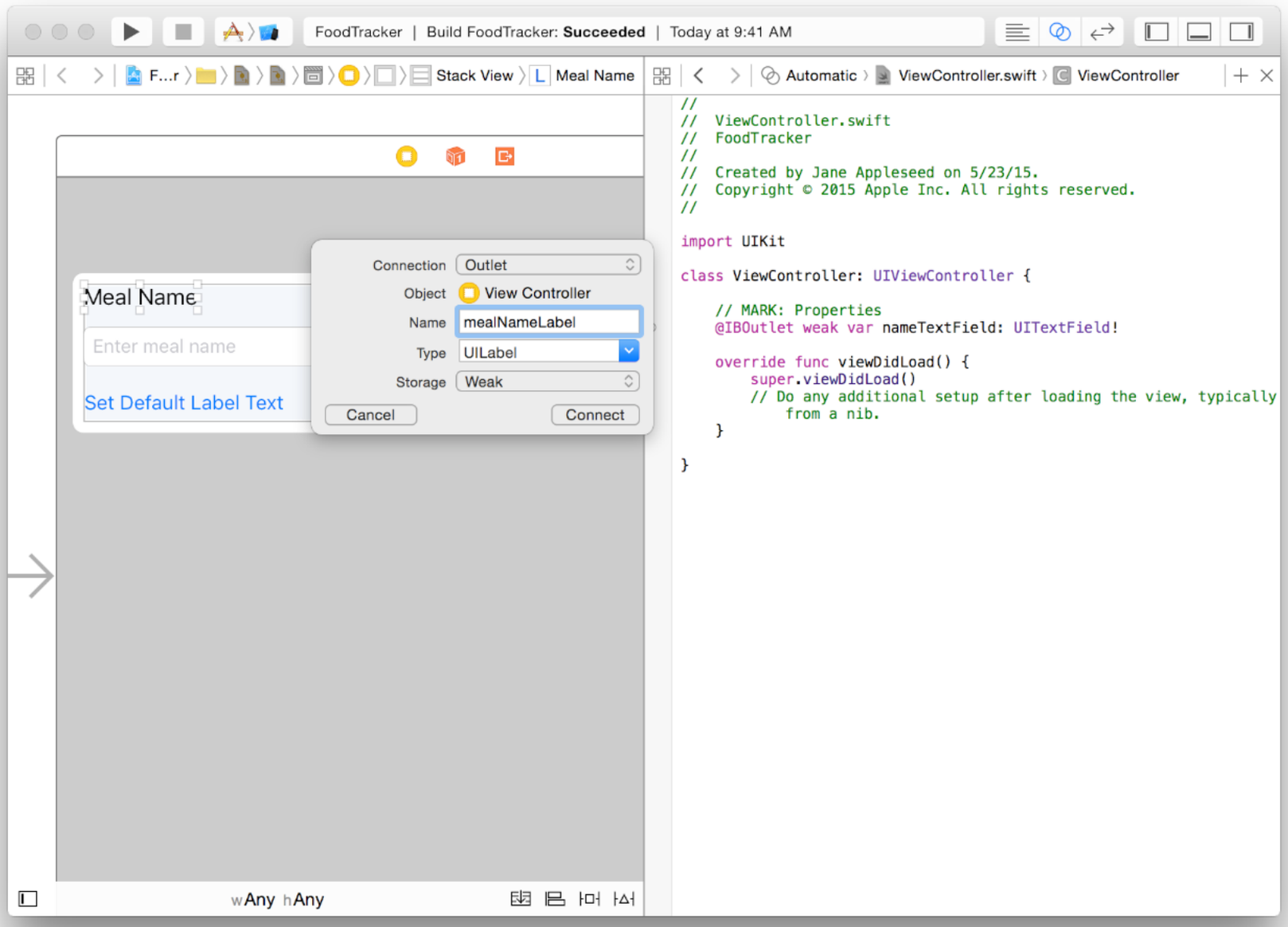
To connect the label to the ViewController.swift code

1. In your storyboard, select the label.
2. Control-drag from the label on your canvas to the code display in the editor on the right, stopping the drag at the line just below your `nameTextField` property in `ViewController.swift`.



3. In the dialog that appears, for Name, type `mealNameLabel`.

   Leave the rest of the options as they are. Your dialog should look like this:



4. Click Connect.

Again, Xcode adds the necessary code to `ViewController.swift` to store a pointer to the label and configures the storyboard to set up that connection. This outlet is similar to the text field, except for its name and its type (which is a `UILabel`, to match the type of object that's in the storyboard).

```
@IBOutlet weak var mealNameLabel: UILabel!
```

Now that you have a way to refer to the interface elements from code, you need to define a user-initiated event that triggers interaction between those elements. That's where actions come in.

Define an Action to Perform

iOS apps are based on event-driven programming. That is, the flow of the app is determined by events: system events and user actions. The user performs actions in the interface that 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 UI. 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.

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

You create an action in the same way you create an outlet: Control-drag from a particular object in your storyboard to a view controller file. This operation creates a method in your view controller file that gets triggered when a user interacts with the object the action method is attached to.

Start by creating a simple action: when a user taps the Set Default Label Text button in your UI, set the label to display a default value, `Default Text`. (The code to set the label to the text in the text field is a bit more involved, so you'll write that in the next section.)

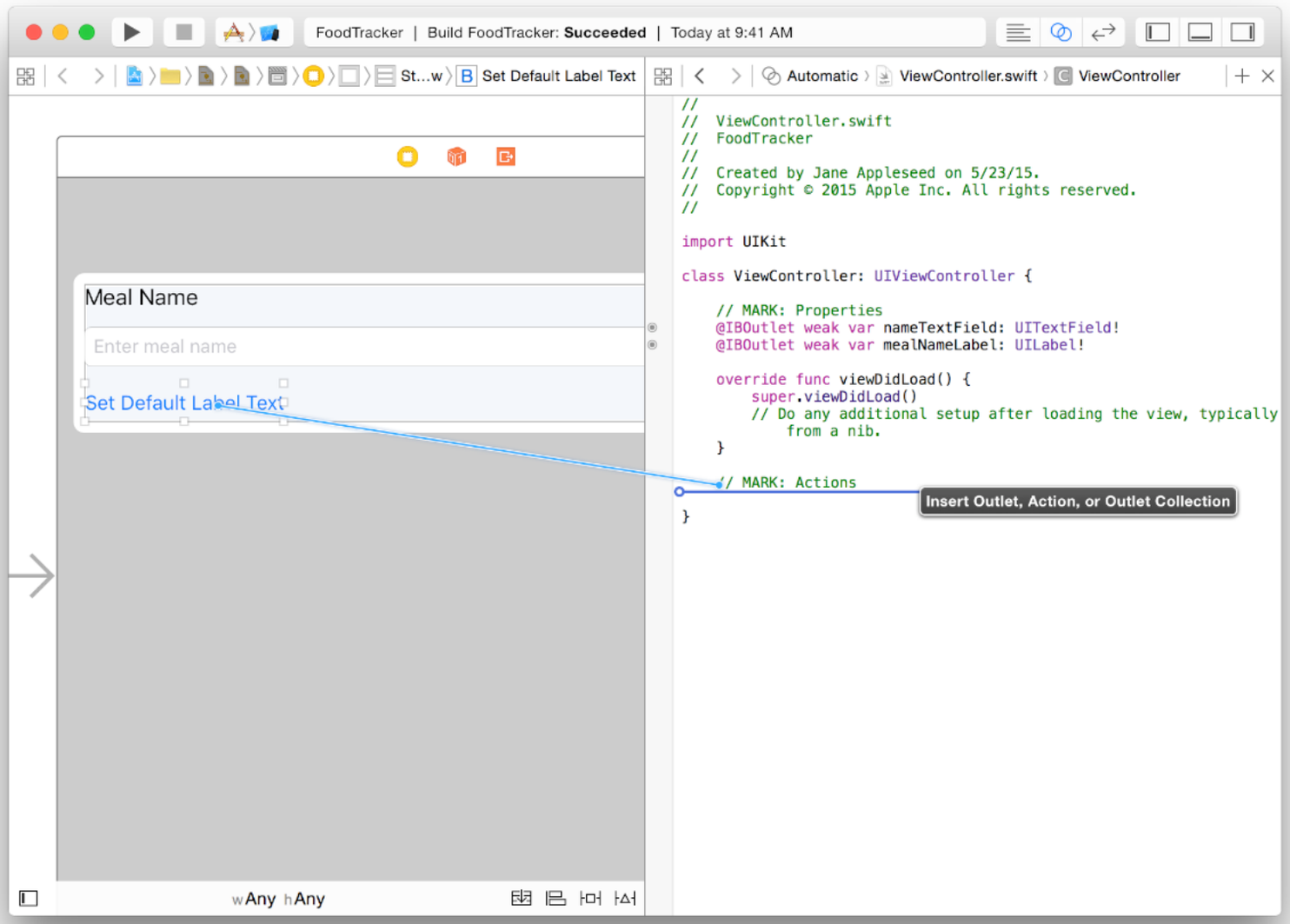
To create a label reset action in the ViewController.swift code

1. In `ViewController.swift`, before the last curly brace (`}`), add the following:

   ```
   // MARK: Actions
   ```

   This comment indicates that this is the section of your code that lists actions.

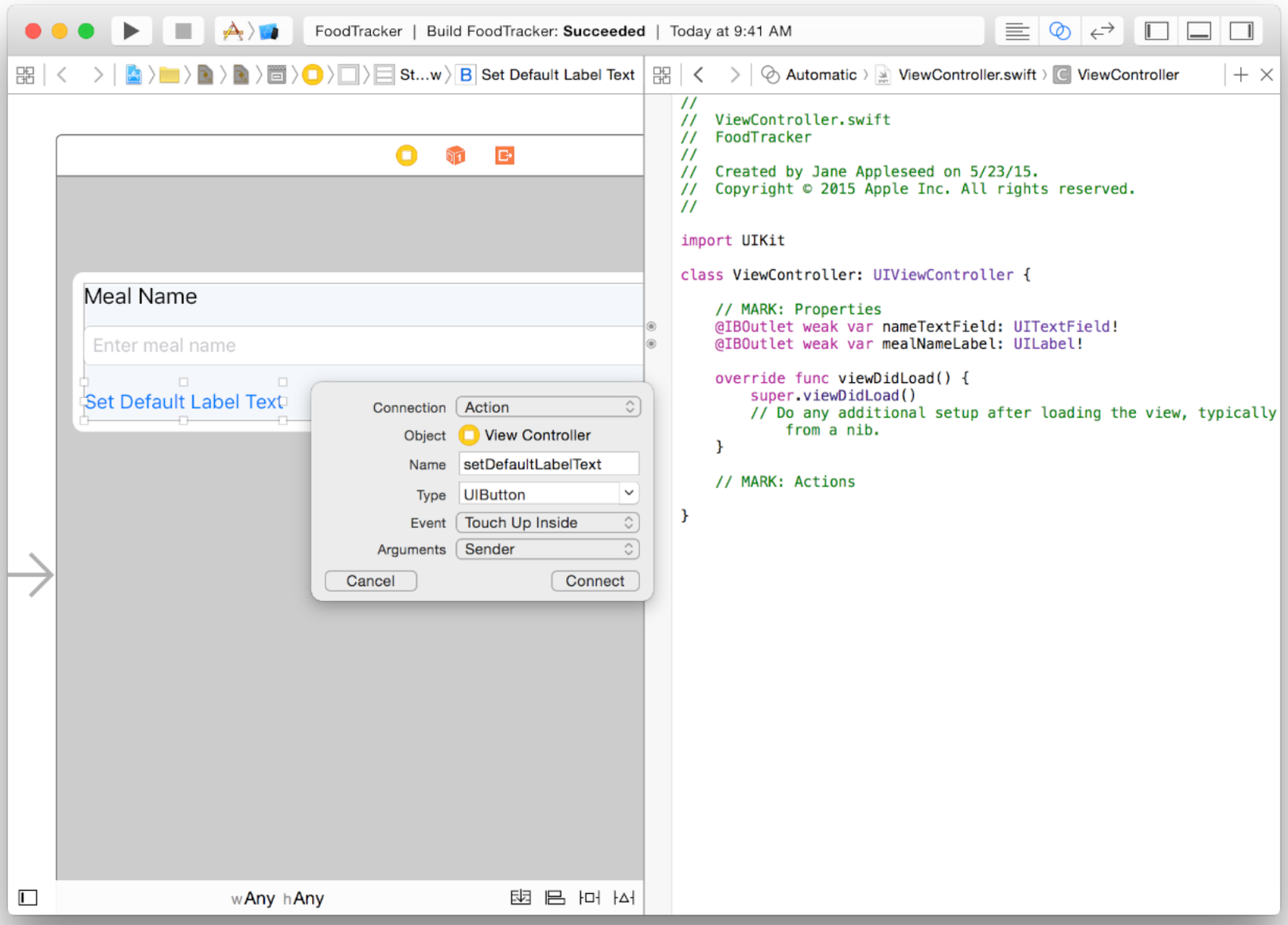
2. In your storyboard, select the Set Default Label Text button.
3. Control-drag from the Set Default Label Text button on your canvas to the code display in the editor on the right, stopping the drag at the line below the comment you just added in `ViewController.swift`.



4. In the dialog that appears, for Connection, select Action.
5. For Name, type `setDefaultLabelText`.
6. For Type, select `UIButton`.

   You may have noticed that the value of the Type field defaults to `AnyObject`. In Swift, `AnyObject` is a type used to describe an object that can belong to any class. Specifying the type of this action method to be `UIButton` means that only button objects can connect to this action. Although this isn't significant for the action you're creating right now, it's important to remember for later.

   Leave the rest of the options as they are. Your dialog should look like this:



7. Click Connect.

Xcode adds the necessary code to `ViewController.swift` to set up the action method.

```
@IBAction func setDefaultLabelText(sender: UIButton) {
}
```

The `sender` parameter points to the object that was responsible for triggering the action—in this case, a button. The `IBAction` attribute indicates that the method is an action that you can connect to from your storyboard in Interface Builder. The rest of the declaration declares a method by the name of `setDefaultLabelText(_:)`.

Right now, the method declaration is empty. The code to reset the value of the label is quite simple.

To implement the label reset action in the ViewController code

1. In `ViewController.swift`, find the `setDefaultLabelText` action method you just added.
2. In the method implementation, between the curly braces (`{}`), add this line of code:

   ```
   mealNameLabel.text = "Default Text"
   ```

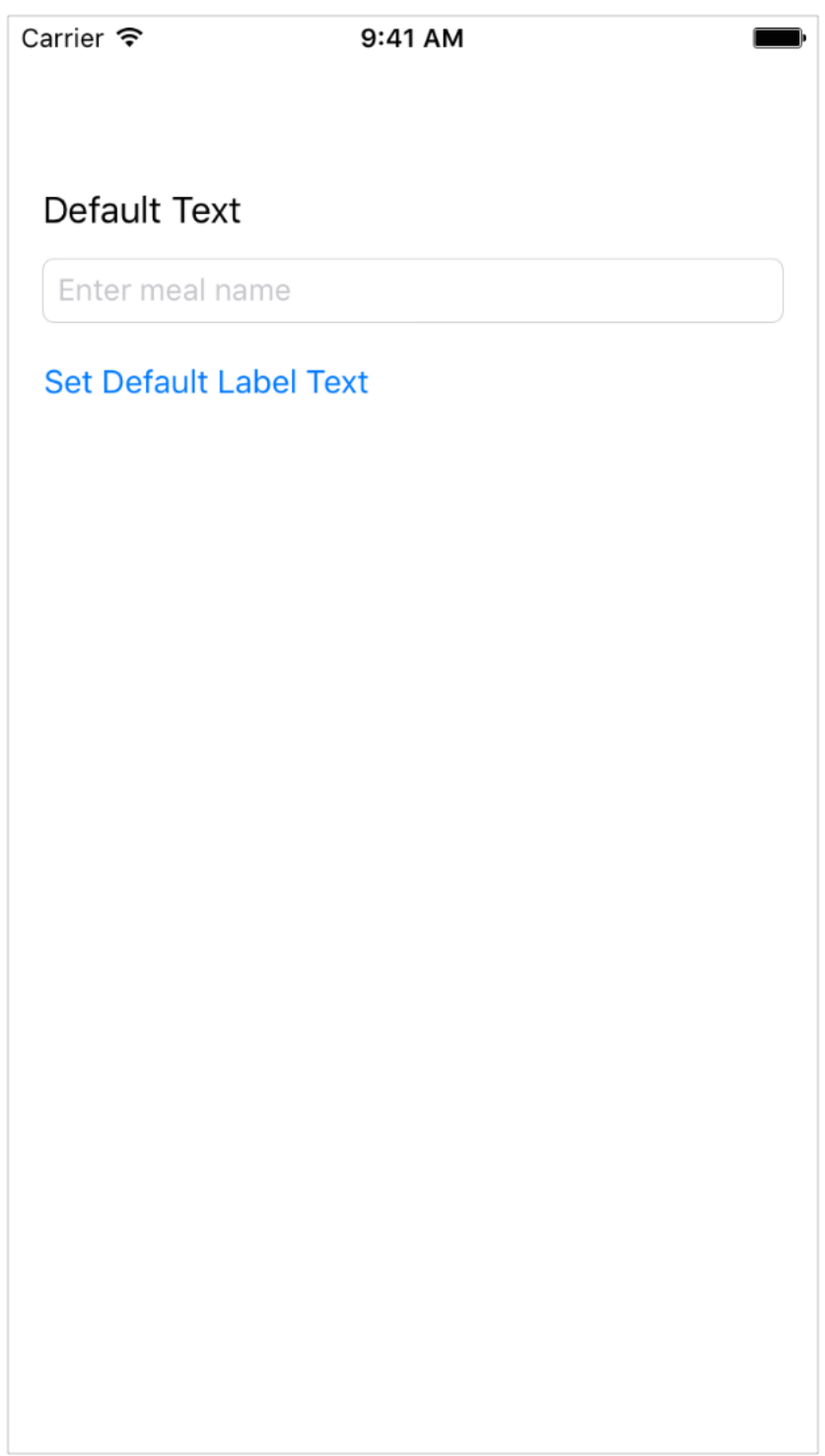
   As you might guess, this code sets the label's `text` property to Default Text.

   Notice that you didn't have to specify the type of Default Text, because Swift's type inference can see that you're assigning to something of type `NSString` and can infer the type correctly.

iOS handles all of the redrawing code for you, so this is actually all the code you need to write for now. Your `setDefaultLabelText(_:)` action method should look like this:

```
@IBAction func setDefaultLabelText(sender: UIButton) {
    mealNameLabel.text = "Default Text"
}
```

Checkpoint: Test your changes by running Simulator. When you click the Set Default Label Text button, the label should change from `Meal Name` (the value set in your storyboard) to `Default Text` (the value set by the action).



The behavior you just implemented is an example of the target-action pattern in iOS app design. Target-action is a design in which one object sends a message to another object when a specific event occurs. In this case, the event is the user tapping the Set Default Label Text button, the action is `setDefaultLabelText`, the target is `ViewController` (where the action method is defined), and the sender is the Set Default Label Text button. The message is an action method defined in source code, and the target—the object that receives the message—is an object capable of performing the action. The object that sends the action message is usually a control—such as a button, slider, or switch—that can trigger an event in response to user interaction such as tap, drag, or value change. This pattern is extremely common in iOS app programming, and you'll be seeing much more of it throughout the rest of the lessons.

Process User Input

At this point, you have a way of resetting the label to a default value, and now you'll add behavior to set it to the value in the text field. To keep things simple, you'll rely on the user's action of tapping the Return button on the text field's keyboard to indicate that the label should update.

When you work with accepting user input from a text field, you need some help from a text field delegate. A delegate is an object that acts on behalf of, or in coordination with, another object. The delegating object—in this case, the text field—keeps a reference to the other object—the delegate—and at the appropriate time, the delegating object sends a message to the delegate. The message tells the delegate about an event that the delegating object is about to handle or has just handled. The delegate may respond by for example, updating the appearance or state of itself or of other objects in the app, or returning a value that affects how an impending event is handled.

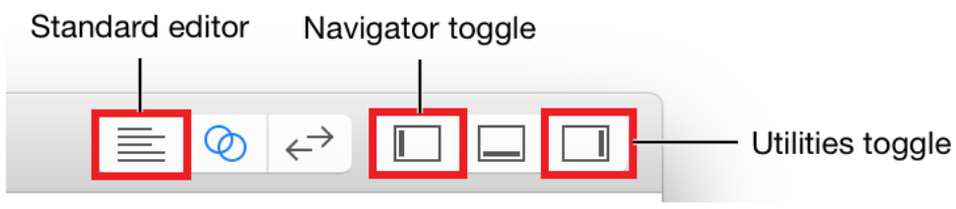
A text field's delegate communicates with the text field while its text is being edited, and knows when important events occur—such as when a user starts or stops editing text. The delegate can use this information to save or clear data at the right time, dismiss the keyboard, and so on.

Any object can serve as a delegate for another object as long as it conforms to the appropriate protocol. The protocol that defines a text field's delegate is called `UITextFieldDelegate`. In this case, because `ViewController` keeps a reference to the text field, you'll make `ViewController` the text field's delegate.

First, you need to have `ViewController` adopt the `UITextFieldDelegate` protocol. You adopt a protocol by listing it as part of the class declaration line.

To adopt the UITextFieldDelegate protocol

1. If the assistant editor is open, return to the standard editor by clicking the Standard button.



   Expand the project navigator and utility area by clicking the Navigator and Utilities buttons in the Xcode toolbar.

2. In the project navigator, select `ViewController.swift`.
3. In `ViewController.swift`, find the `class` line, which should look like this:

```
class ViewController: UIViewController {
```

4. After `UIViewController`, add a comma (`,`) and `UITextFieldDelegate` to adopt the protocol.

```
class ViewController: UIViewController, UITextFieldDelegate {
```

By adopting the protocol, you gave the `ViewController` class the ability to identify itself as a `UITextFieldDelegate`. This means you can set it as the delegate of the text field and implement some of its behavior to handle the text field's user input.

To set ViewController as the delegate for nameTextField

1. In `ViewController.swift`, find the `viewDidLoad()` method, which should look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.
}
```

   The template implementation of this method includes a comment. You don't need this comment in your method implementation, so go ahead and delete it.

2. Below the `super.viewDidLoad()` line, add a blank line and the following:

```
// Handle the text field's user input through delegate callbacks.
nameTextField.delegate = self
```

   The `self` refers to the `ViewController` class, because it's referenced inside the scope of the `ViewController` class definition.

   You can add your own comments to help you understand what's happening in your code.

Your `viewDidLoad()` method should look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()

    // Handle the text field's user input through delegate callbacks.
    nameTextField.delegate = self
}
```

`ViewController` is now a delegate for `nameTextField`.

The `UITextFieldDelegate` protocol contains optional methods, which means that you're not required to implement them. But to get the specific behavior you want, you'll need to implement two of these methods for now:

```
func textFieldShouldReturn(textField: UITextField) -> Bool
func textFieldDidEndEditing(textField: UITextField)
```

To understand when these methods get called and what they need to do, it's important to know how text fields respond to user events. When the user taps a text field, it automatically becomes first responder. In an app, the first responder is an object that is first on the line for receiving many kinds of app events, including key events, motion events, and action messages, among others. In other words, many of the events generated by the user are initially routed to the first responder.

As a result of the text field becoming first responder, iOS displays the keyboard and begins an editing session for that text field. What a user types using that keyboard gets inserted into the text field.

When a user wants to finish editing the text field, the text field needs to resign its first-responder status. Because the text field will no longer be the active object in the app, events need to get routed to a more appropriate object.

This is where your implementation of `UITextFieldDelegate` methods comes in. You need to specify that the text field should resign its first-responder status when the user taps a button to end editing in the text field. You do this in the `textFieldShouldReturn(_:)` method, which gets called when the user taps Return (or in this case, Done) on the keyboard.

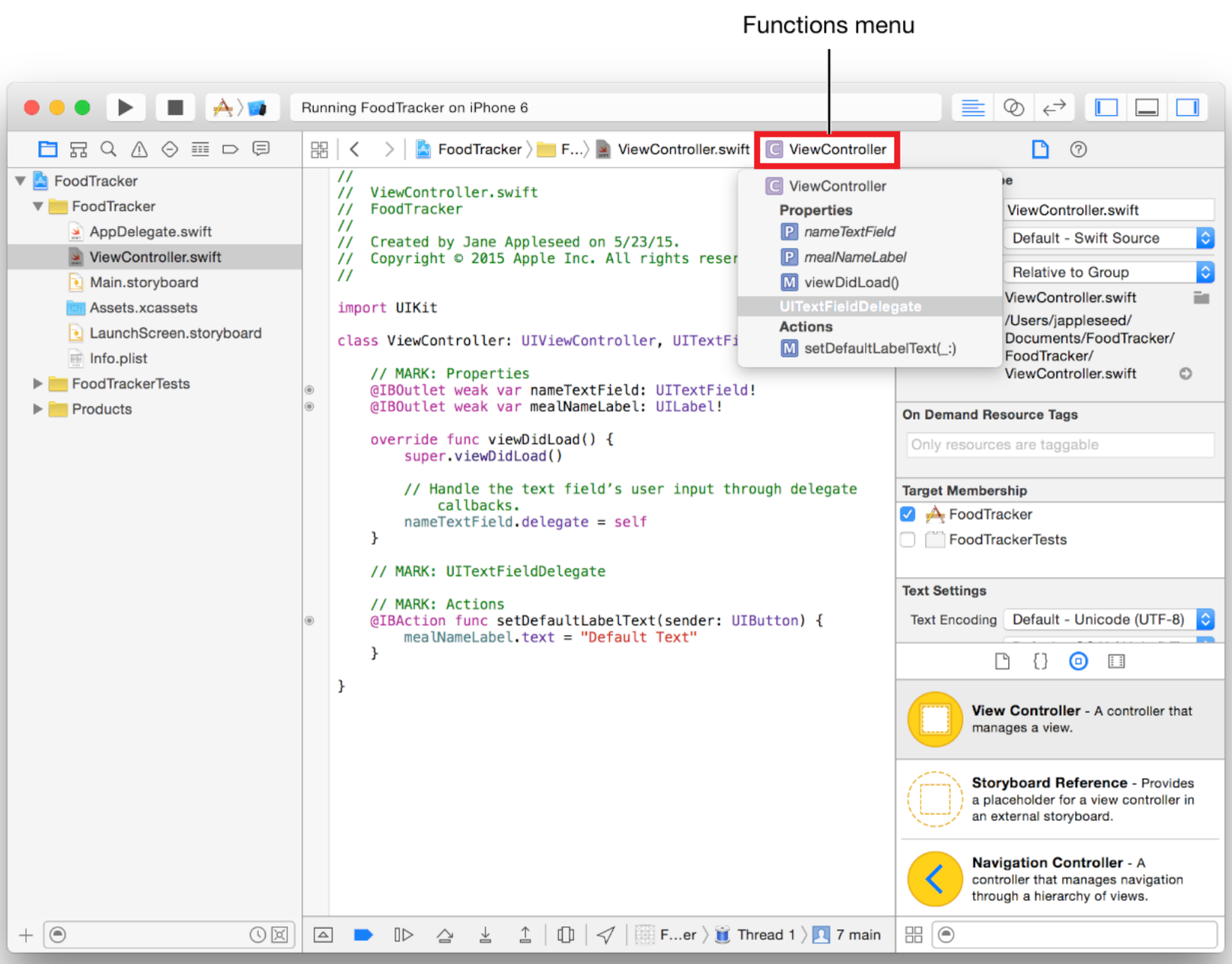
To implement the UITextFieldDelegate protocol method textFieldShouldReturn(_:)

1. In `ViewController.swift`, right above the `// MARK: Actions` section, add the following:

    ```
    // MARK: UITextFieldDelegate
    ```

    This comment is used to organize your code and to help you (and anybody else who reads your code) navigate through it.

    You've added several of these comments so far. Xcode lists each of these comments as a section title in the source code file's functions menu, which appears if you click the name of the file at the top of the editor area. The functions menu lets you jump to a section in your code quickly. You'll notice the sections you denoted by `// MARK:` listed here. You can click on one of the section titles to jump to that section in the file.

2. Below the comment, add the following method:

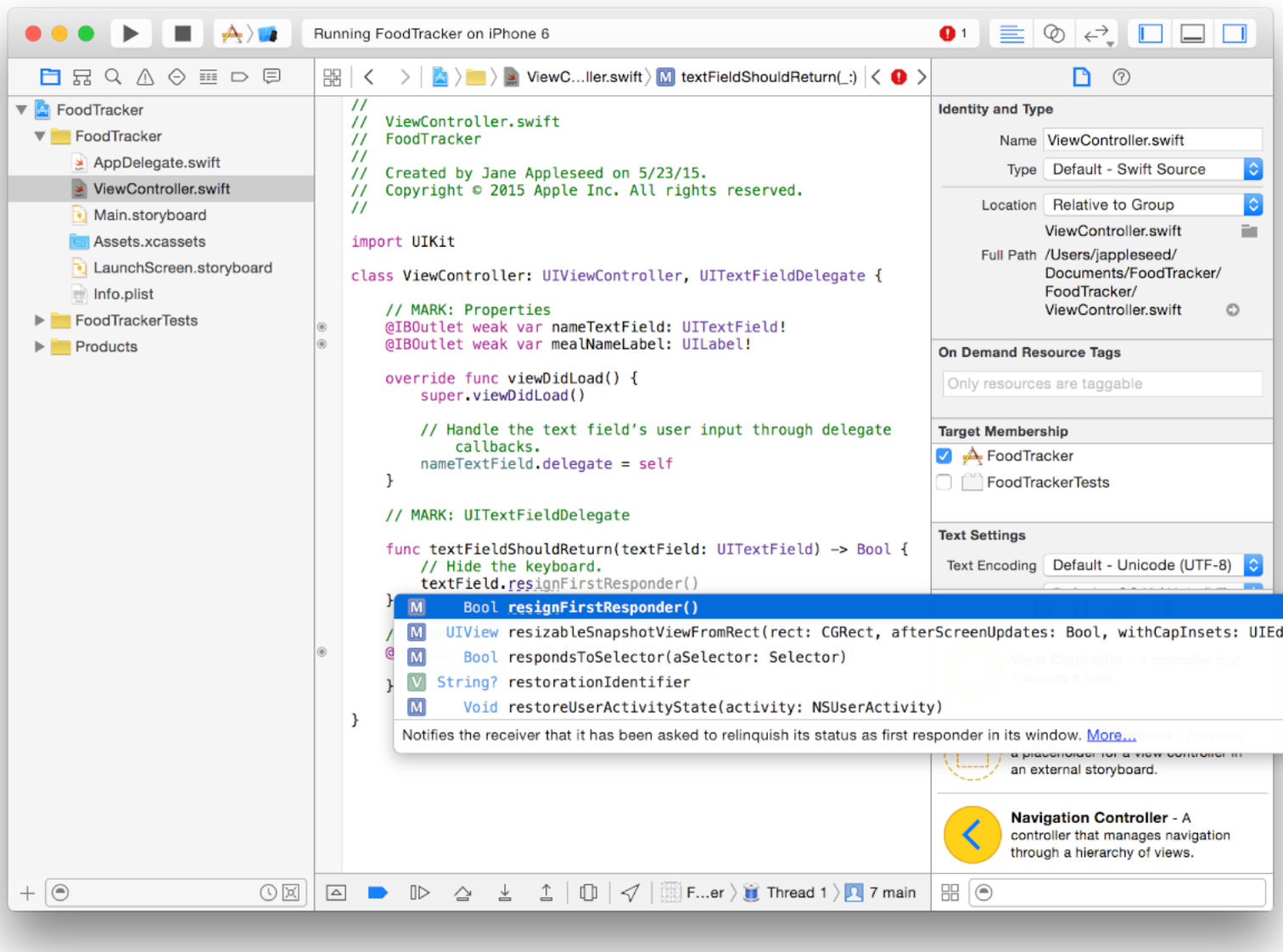
    ```
    func textFieldShouldReturn(textField: UITextField) -> Bool {
    }
    ```

3. In this method, add the following code to resign the text field's first-responder status, and a comment to

describe what the code does:

```
// Hide the keyboard.
textField.resignFirstResponder()
```

Try typing the second line instead of just copying and pasting. You'll find that code completion is one of the great time-saving features of Xcode. When Xcode brings up the list of potential completions, scroll through the list until you find the one you want and then press Return. Xcode inserts the whole line for you.



4. In this method, add the following line of code:

    ```
    return true
    ```

    Because this method returns a Boolean value, returning the value `true` indicates that the text field should respond to the user pressing the Return key by dismissing the keyboard.

Your `textFieldShouldReturn(_:)` method should look like this:

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
    // Hide the keyboard.
    textField.resignFirstResponder()
    return true
}
```

The second method that you need to implement, `textFieldDidEndEditing(_:)`, is called after the text field resigns its first-responder status. This method will be called after the `textFieldShouldReturn` method you just implemented.

The `textFieldDidEndEditing(_:)` method gives you a chance to read the information entered into the text field and do something with it. In your case, you'll take the text that's in the text field and use it to change the value of the label in your UI.

To implement the UITextFieldDelegate protocol method textFieldDidEndEditing(_:)

1. In `ViewController.swift`, after the `textFieldShouldReturn(_:)` method, add the following method:

    ```
    func textFieldDidEndEditing(textField: UITextField) {
    }
    ```

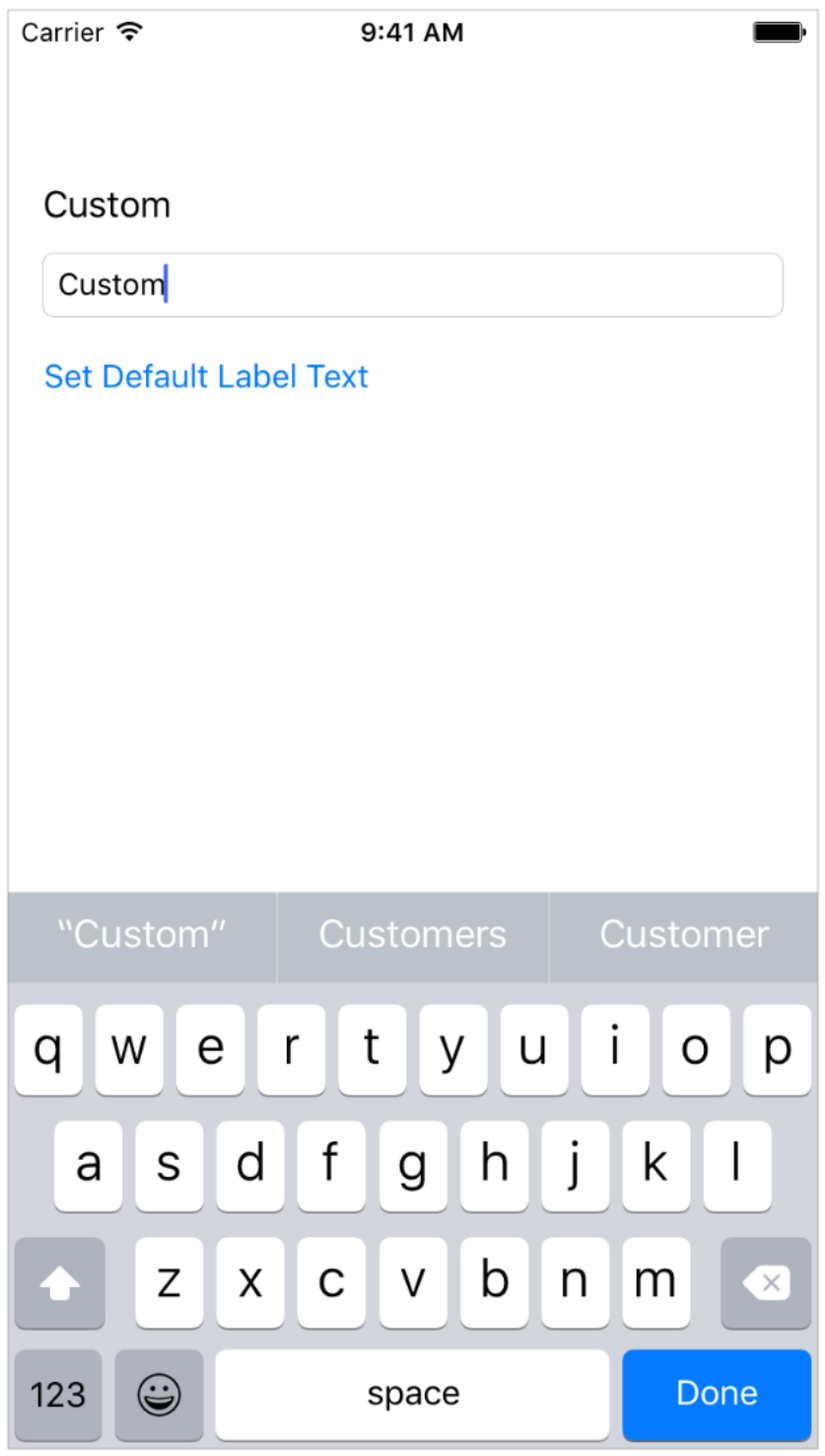
2. In this method, add the following line of code:

```
mealNameLabel.text = textField.text
```

That's all you need to do to see the result. Your `textFieldDidEndEditing(_:)` method should look like this:

```
1  func textFieldDidEndEditing(textField: UITextField) {
2      mealNameLabel.text = textField.text
3  }
```

Checkpoint: Test your changes by running Simulator. You can select the text field and type text into it. When you click the Done button on the keyboard, the keyboard is dismissed and the label text changes to display the text in the text field. When you click the Set Default Label Text button, the label changes from what's currently displayed in the label to `Default Text` (the value set by the action you defined earlier).



NOTE

To see the completed sample project for this lesson, download the file and view it in Xcode.

Download File