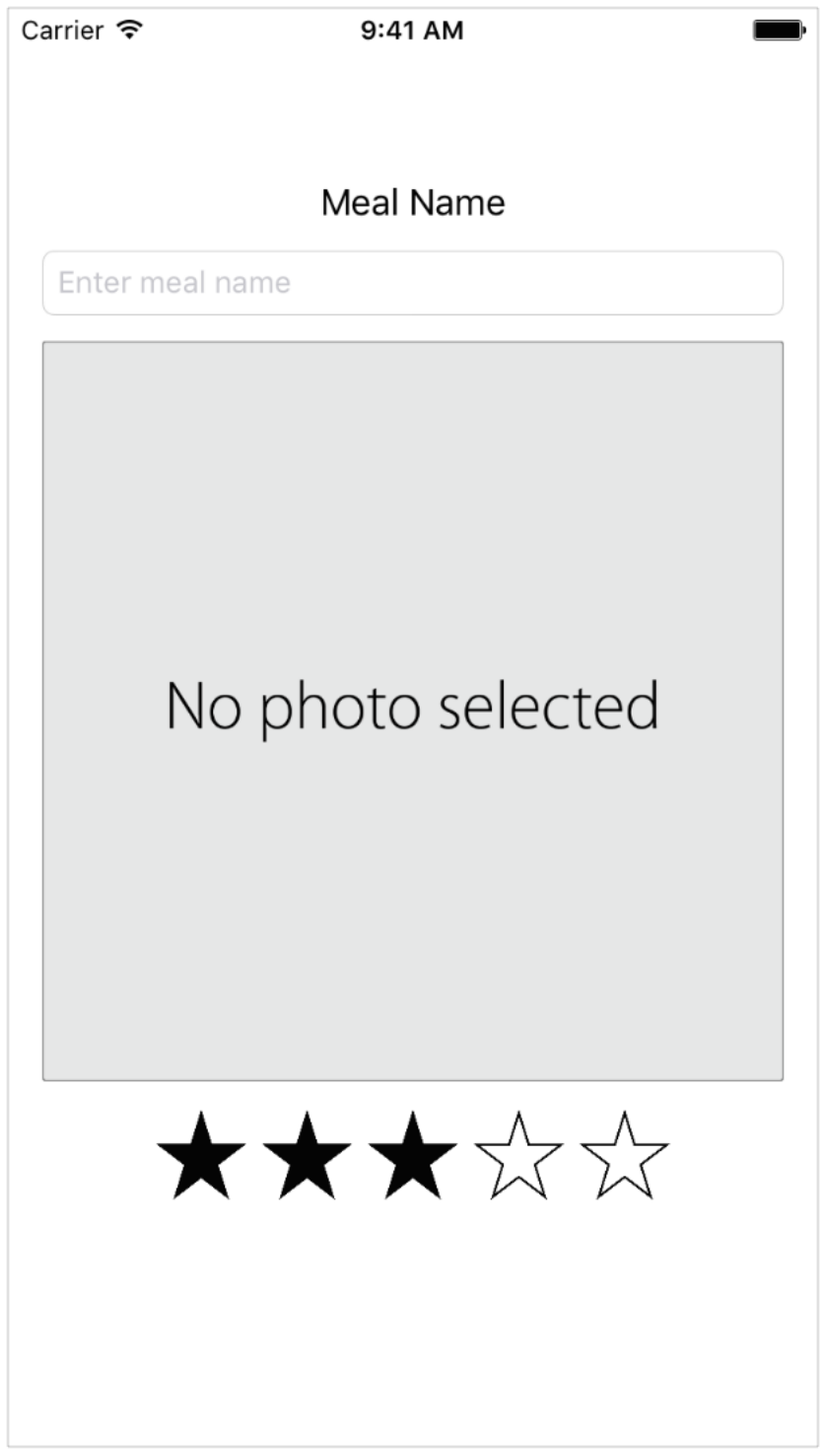
Implement a Custom Control

In this lesson, you'll implement a rating control for the FoodTracker app. When you're finished, your app will look something like this:



Learning Objectives

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

- Create and associate custom source code files with elements in a storyboard
- Define a custom class
- Implement an initializer on a custom class
- Use `UIView` as a container
- Understand how to display views programmatically

Create a Custom View

To be able to rate a meal, users need a control that lets them select the number of stars they want to assign to the meal. There are many ways to implement this, but you'll focus on one that involves creating a custom view that you define in code and use in your storyboard.

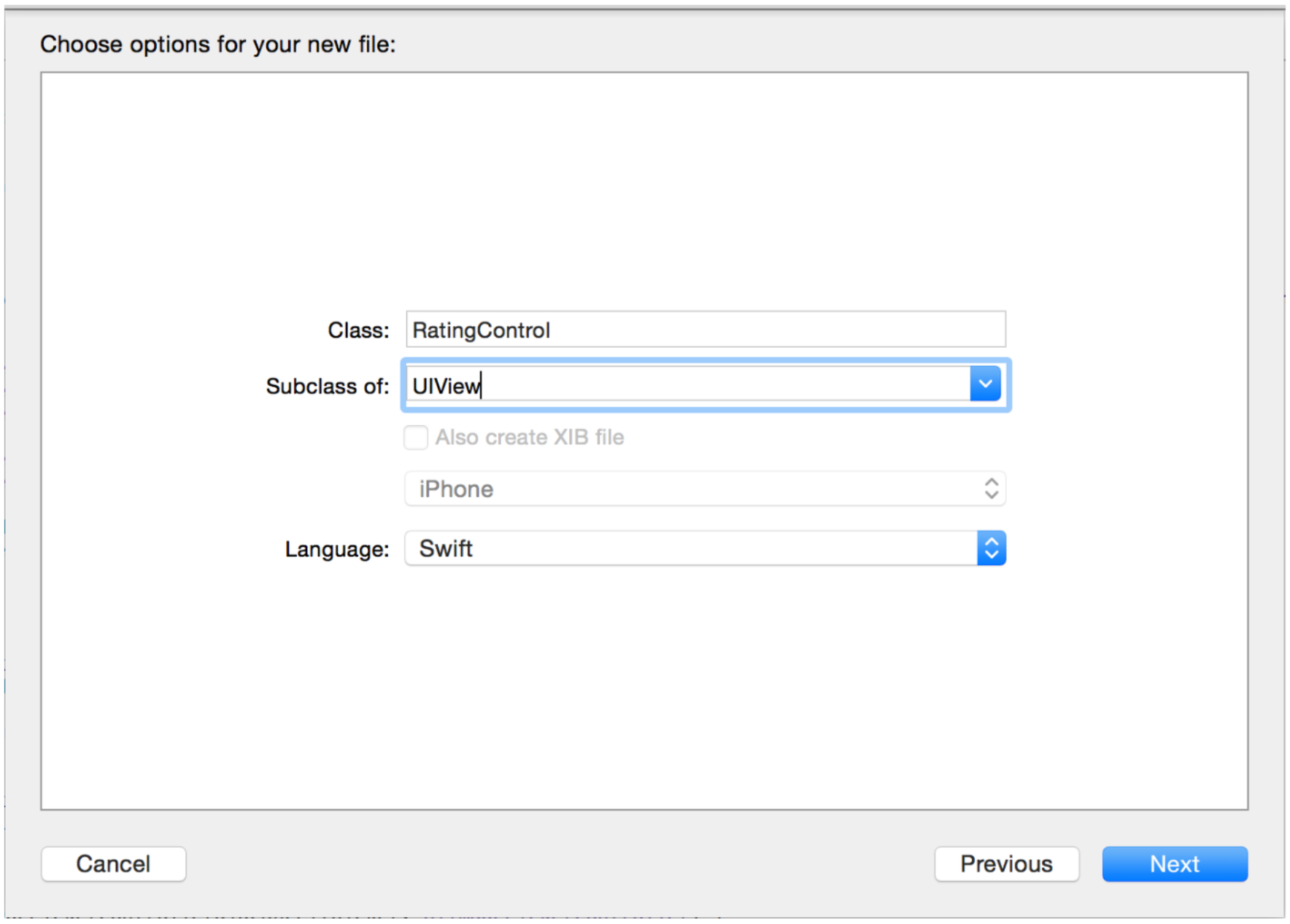
Here's the rating control you're implementing:

The rating control will let users choose 0, 1, 2, 3, 4, or 5 stars for a meal. When a user taps a star, all stars leading up to and including that star (from the left) are filled in. A filled-in star counts as a rating; an empty star doesn't.

To begin designing the UI, interaction, and behavior of this control, start by creating a custom view (`UIView`) subclass.

To create a subclass of UIView

1. Choose File > New > File (or press Command-N).
2. On the left of the dialog that appears, select Source under iOS.
3. Select Cocoa Touch Class, and click Next.
4. In the Class field, type `RatingControl`.
5. In the "Subclass of" field, select `UIView`.
6. Make sure the Language option is set to Swift.



7. Click Next.

   The save location defaults to your project directory.

   The Group option defaults to your app name, FoodTracker.

   In the Targets section, your app is selected and the tests for your app are unselected.
8. Leave these defaults as they are, and click Create.

   Xcode creates a file that defines the `RatingControl` class: `RatingControl.swift`. `RatingControl` is a custom view subclass of `UIView`.
9. In `RatingControl.swift`, delete the comments that come with the template implementation so you can start working with a blank slate.

   The implementation should look like this:

```
import UIKit

class RatingControl: UIView {

}
```

You typically create a view in one of two ways: by initializing the view with a frame so that you can manually add the view to your UI, or by allowing the view to be loaded by the storyboard. There's a corresponding

initializer for each approach: `init(frame:)` for the frame and `init?(coder:)` for the storyboard. Recall that an initializer is a method that prepares an instance of a class for use, which involves setting an initial value for each property and performing any other setup.

Because you'll be using this view in your storyboard, start by overriding its superclass's implementation of the `init?(coder:)` initializer.

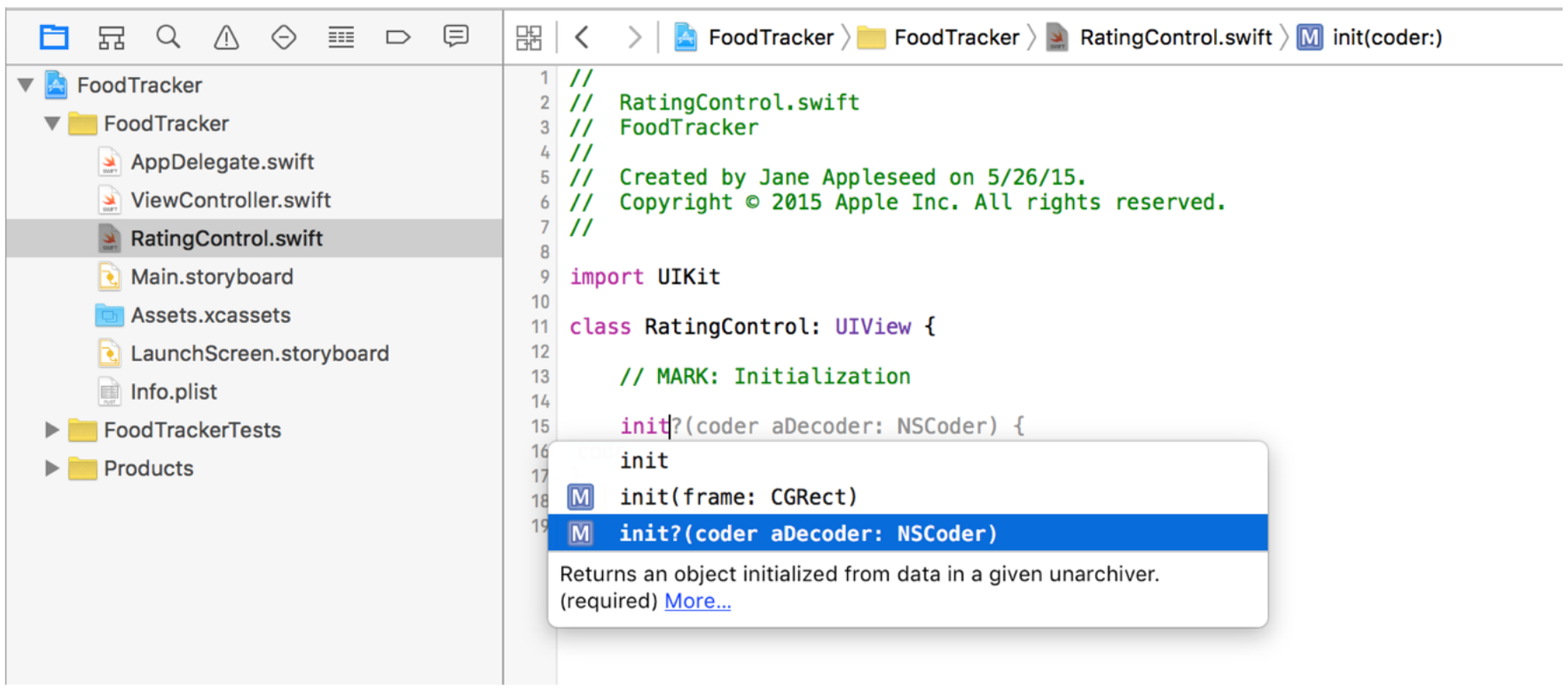
To override the initializer

1. In `RatingControl.swift`, under the `class` line, add this comment.

```
// MARK: Initialization
```

2. Below the comment, start typing `init`.

   The code completion overlay shows up.

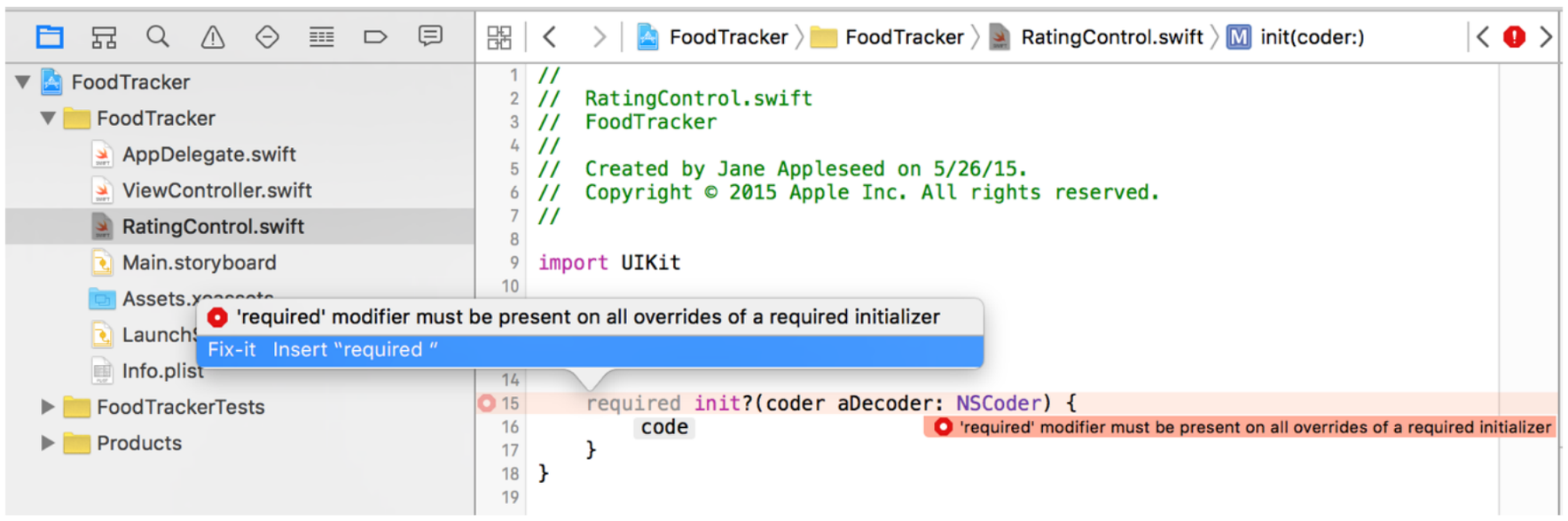


3. Select the third method in the list to get the `init?(coder:)` initializer, and press Return.

```
init?(coder aDecoder: NSCoder) {
}
```

   Xcode inserts the initializer skeleton for you.

4. Click the error fix-it to include the `required` keyword.




```
required init?(coder aDecoder: NSCoder) {
}
```

   Every `UIView` subclass that implements an initializer must include an implementation of `init?(coder:)`. The Swift compiler knows this, and offers a fix-it to make this change in your code. Fix-its are provided by the compiler as potential solutions to errors in your code.

5. Add this line to call the superclass's initializer.

```
super.init(coder: aDecoder)
```

Your `init?(coder:)` initializer should look like this:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}
```

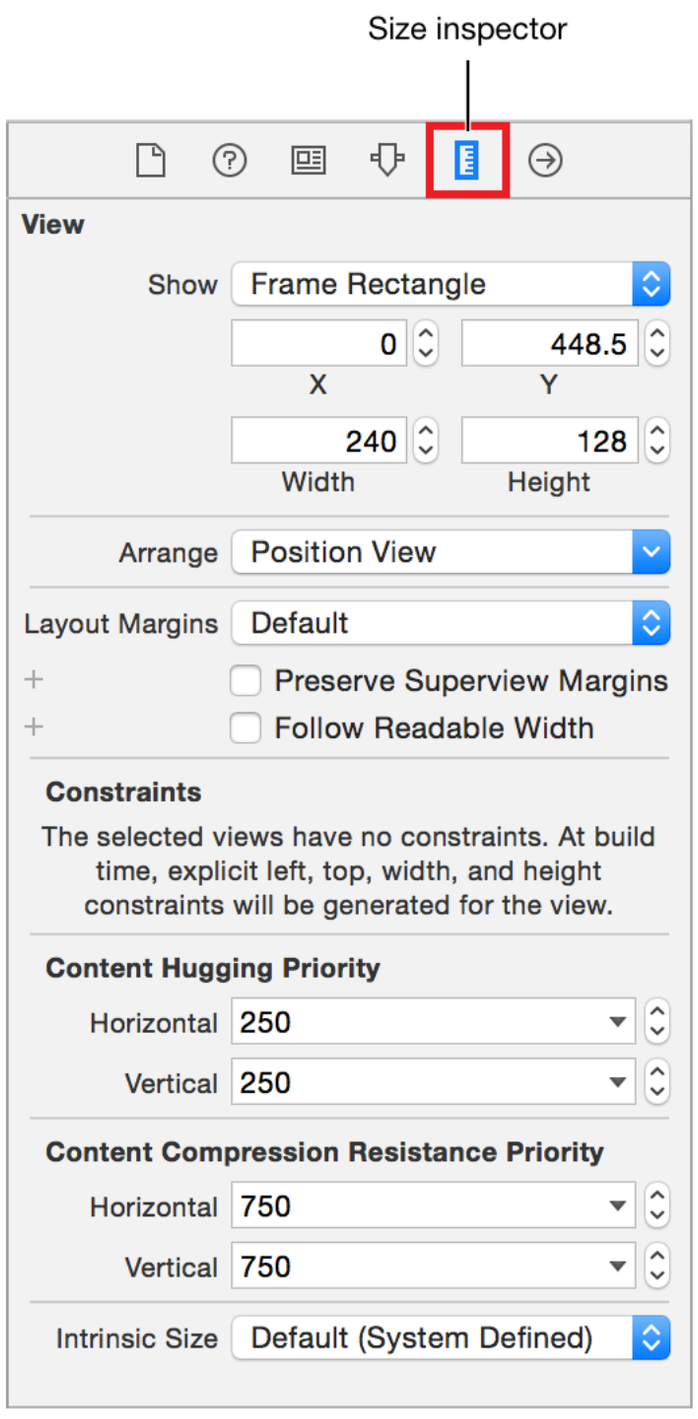
Display the Custom View

To display your custom view, you need to add a view to your UI and establish a connection between that view and the code you just wrote.

To display the view

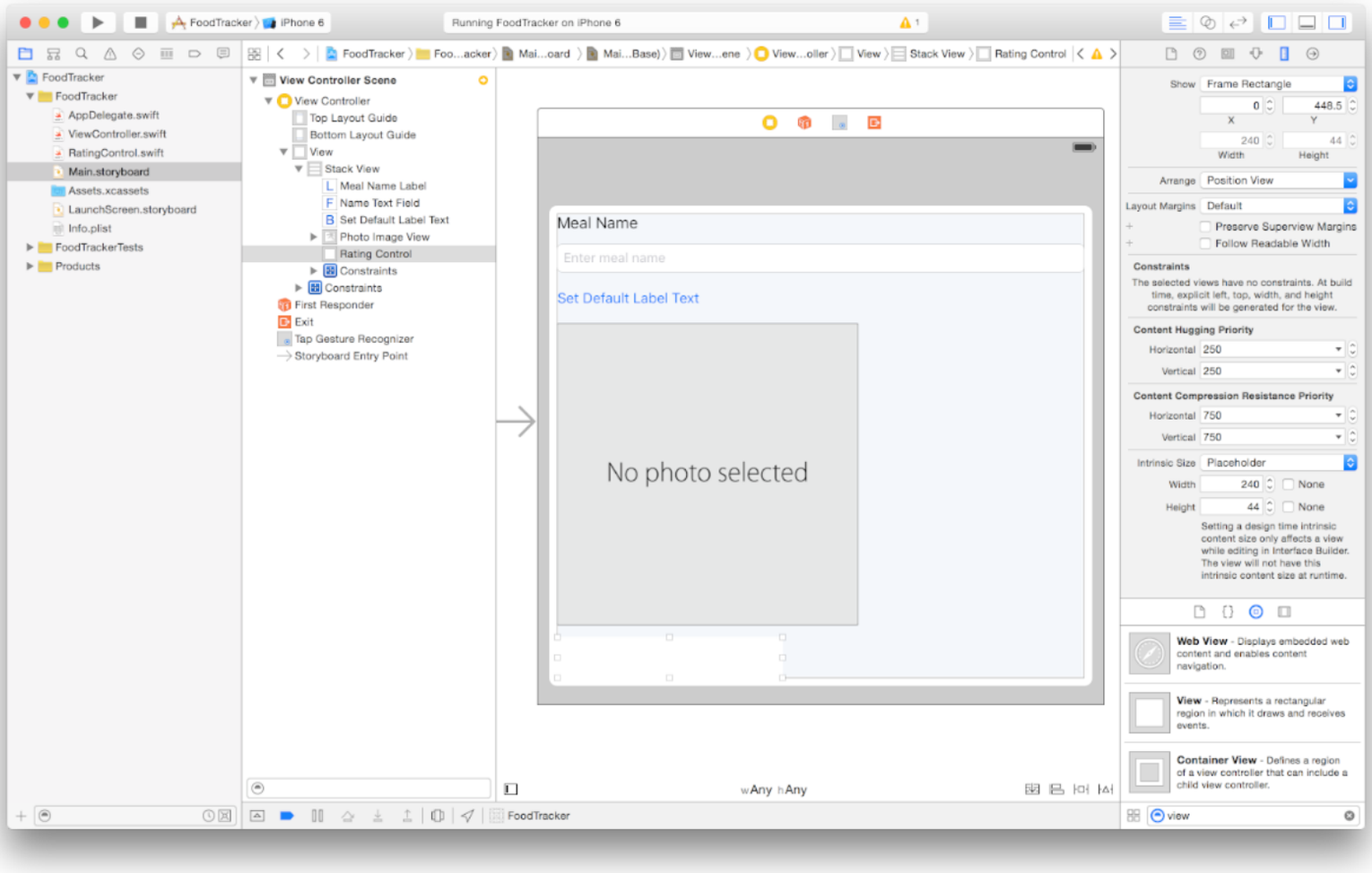
1. Open your storyboard.
2. In your storyboard, use the Object library to find a View object and drag one into your storyboard scene so that it's in the stack view below the image view.
3. With the view selected, open the Size inspector in the utility area.

   Recall that Size inspector appears when you select the fifth button from the left in the inspector selector bar. It lets you edit the size and position of an object in your storyboard.



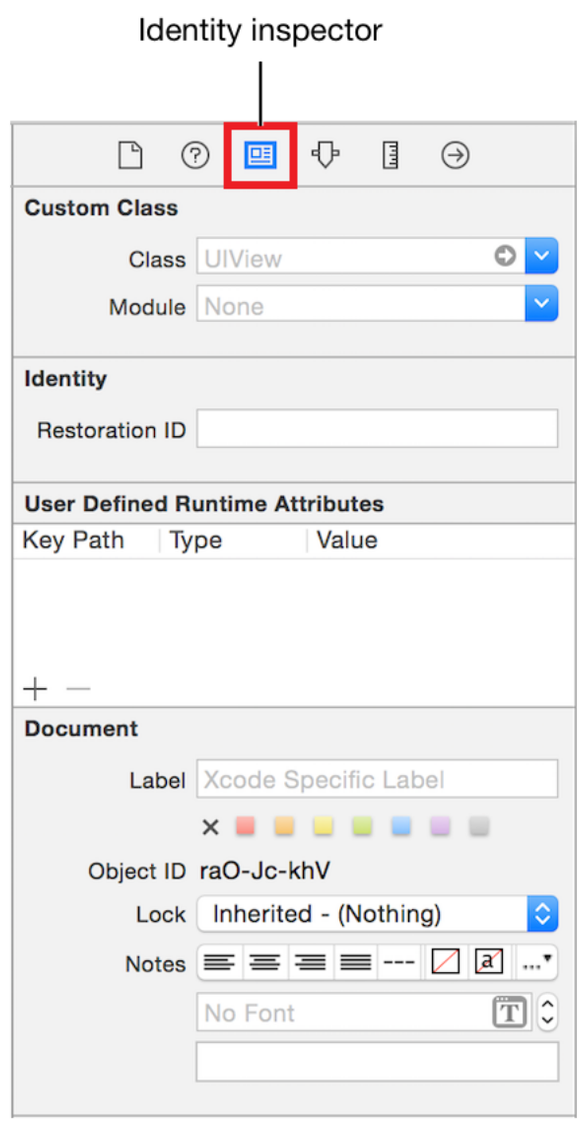
4. In the pop-up menu labeled Intrinsic Size, choose Placeholder. (This field is at the bottom of the Size inspector, so you'll need to scroll down to it.)
5. Type `44` into the Height field and `240` into the Width field located below Intrinsic Size. Press Return.

   Your UI should look something like this:

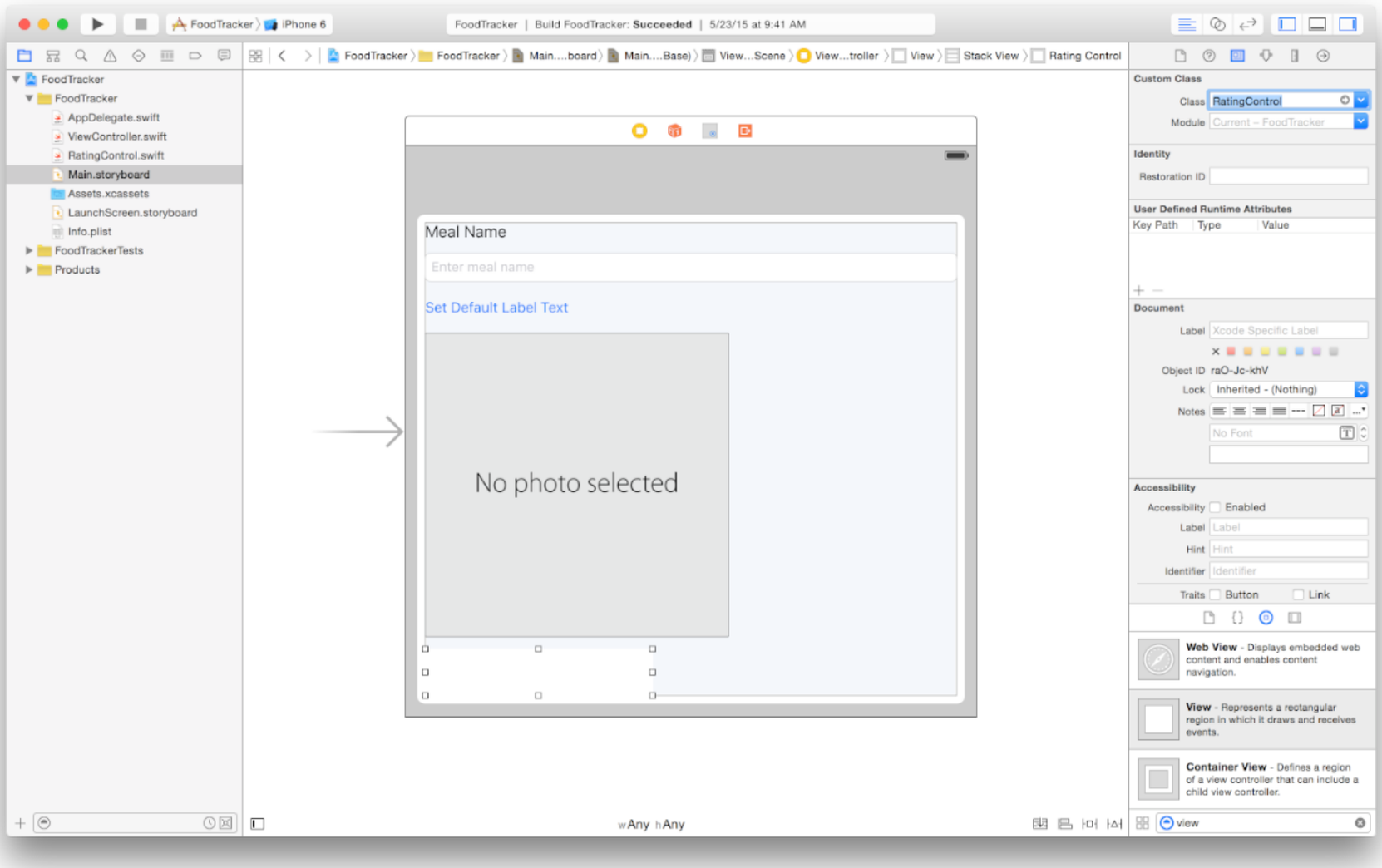


6. With the view selected, open the Identity inspector .

   Recall that 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.



7. In the Identity inspector, find the field labeled Class and select `RatingControl`.



Add Buttons to the View

At this point, you've got the basics of a custom `UIView` subclass, called `RatingControl`. The next thing you'll need to do is add buttons to your view to allow the user to select a rating. Start with something simple, like getting a single red button to show up in your view.

To create a button in your view

1. In the `init?(coder:)` initializer, add the following lines of code to create a red button:

   ```
   let button = UIButton(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
   button.backgroundColor = UIColor.redColor()
   ```

   You're using `redColor()` so it's easy to see where the view is. If you'd like, use one of the other predefined `UIColor` values instead, like `blueColor()` or `greenColor()`.

2. Below the last line, add this code:

   ```
   addSubview(button)
   ```

   The `addSubview()` method adds the button you created to the `RatingControl` view.

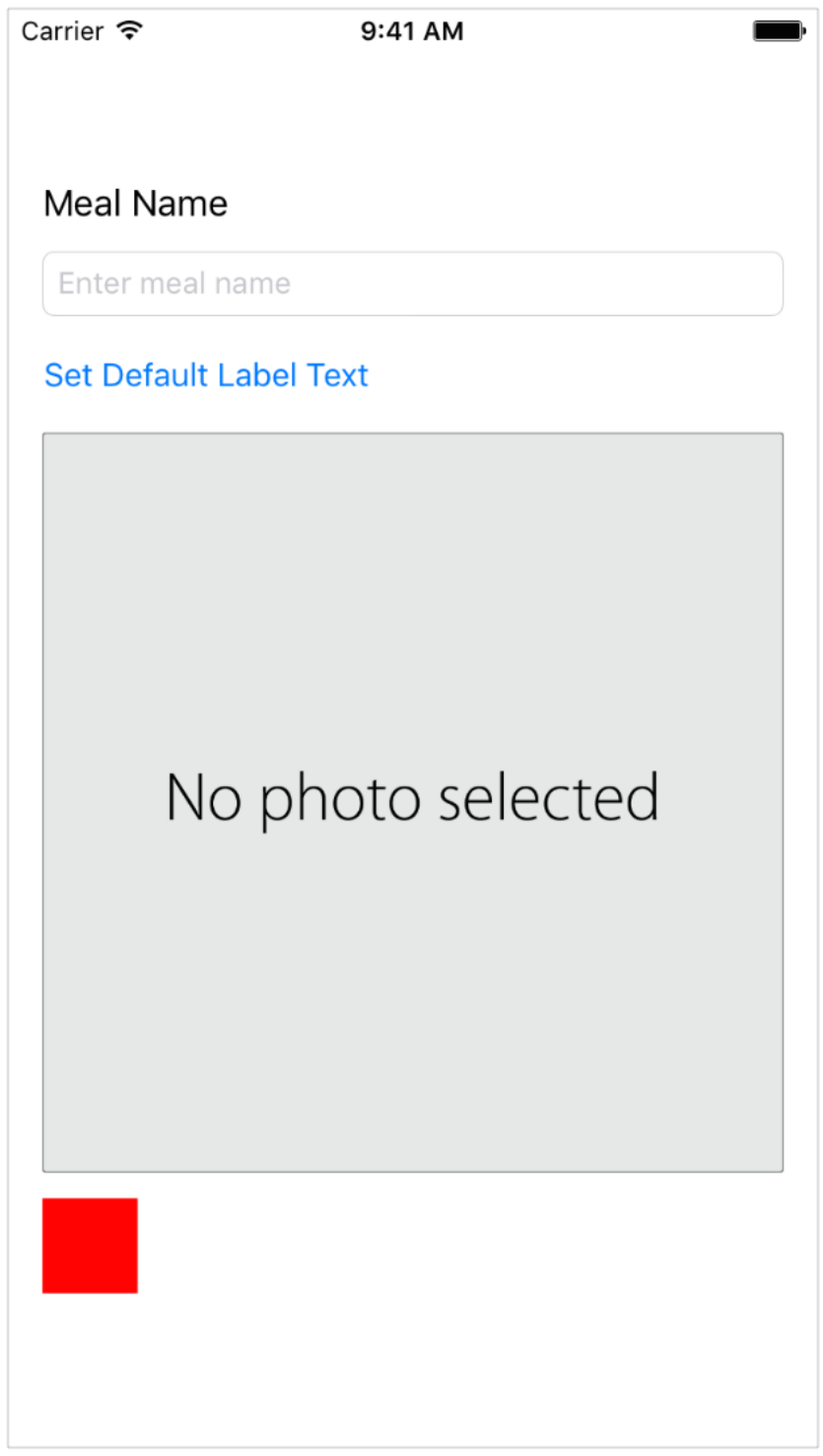
Your `init?(coder:)` initializer should look like this:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let button = UIButton(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
    button.backgroundColor = UIColor.redColor()
    addSubview(button)
}
```

To tell the stack view how to lay out your button, you also need to provide an intrinsic content size for it. To do this, override the `intrinsicContentSize` method to match the size you specified in Interface Builder like this:

```
override func intrinsicContentSize() -> CGSize {
    return CGSize(width: 240, height: 44)
}
```

Checkpoint: Run your app. You should be able to see a view with a small red square inside of it. The red square is the button you added in the initializer.



You'll need this button and eventually the other buttons in this view to perform an action when they're tapped. That action is changing a meal's rating.

To add an action to the button

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

```
// MARK: Button Action
```

2. Under the comment, add the following:

```
func ratingButtonTapped(button: UIButton) {
    print("Button pressed 👍")
}
```

   For now, use the `print()` function to check that the `ratingButtonTapped(_:)` action is linked to the button as expected. This function prints a message to the standard output, which in this case is the Xcode debug console. The console is a useful debugging mechanism that appears at the bottom of the editor area.

   You'll replace this debugging implementation with a real implementation in a little while.

3. Find the `init?(coder:)` initializer:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let button = UIButton(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
    button.backgroundColor = UIColor.redColor()
    addSubview(button)
}
```

4. Before the `addSubview(button)` line, add this:

```
button.addTarget(self, action:
  #selector(RatingControl.ratingButtonTapped(_:)), forControlEvents:
  .TouchDown)
```

You're familiar with the target-action pattern because you've used it to link elements in your storyboard to action methods in your code. Above, you're doing the same thing, except you're creating the connection in code. You're attaching the `ratingButtonTapped(_:)` action method to the `button` object, which will be triggered whenever the `.TouchDown` event occurs. This event signifies that the user has pressed on a button. You set the target to `self`, which in this case is the `RatingControl` class, because that's where the action is defined.

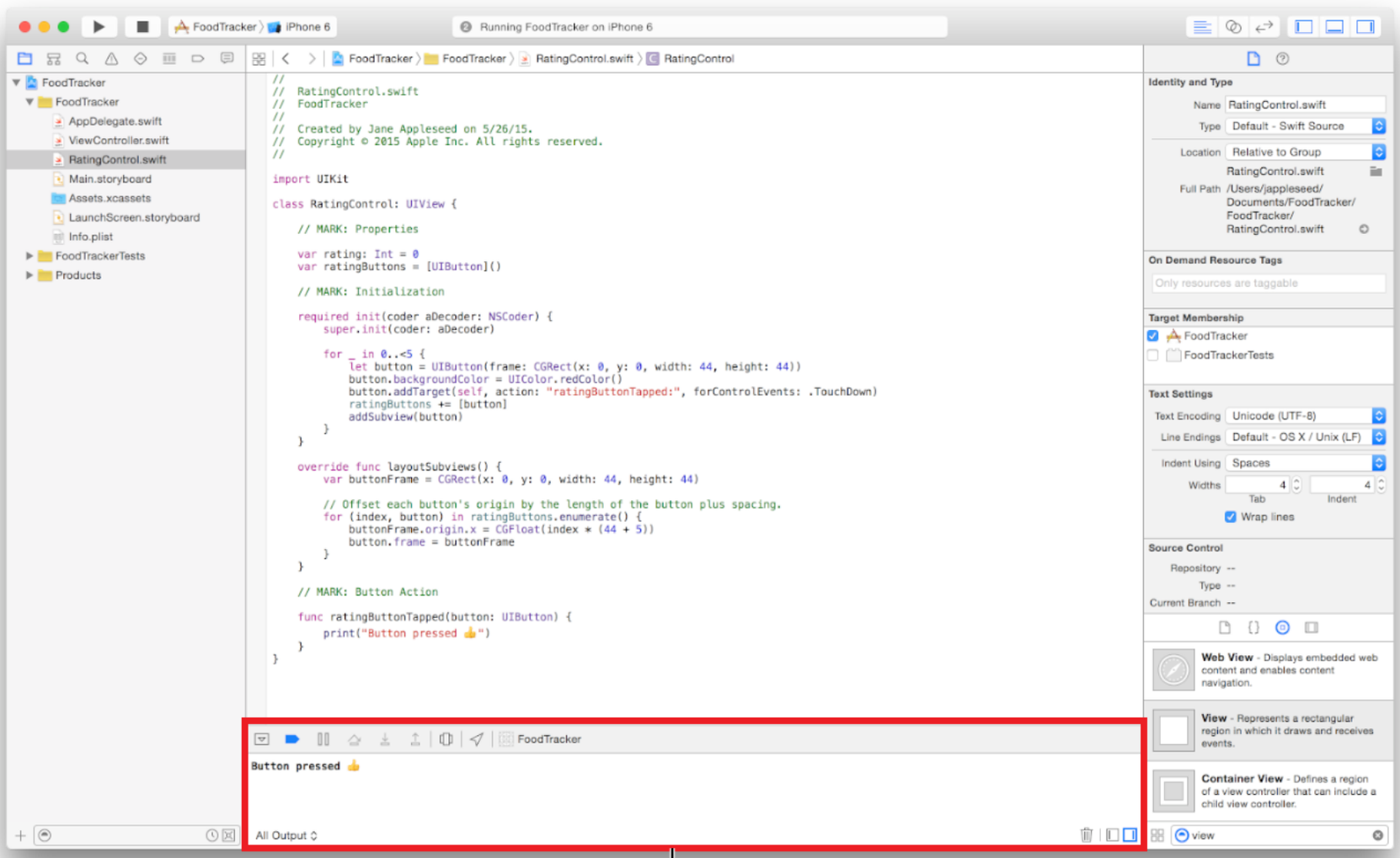
The `#selector` expression returns the `Selector` value for the provided method. A selector is an opaque value that identifies the method. Older APIs used selectors to dynamically invoke methods at runtime. While newer APIs have largely replaced selectors with blocks, many older methods—like `performSelector(_:)` and `addTarget(_:action:forControlEvents:)`—still take selectors as arguments. In this example, the `#selector(RatingControl.ratingButtonTapped(_:))` expression returns the selector for your `ratingButtonTapped(_:)` action method. This lets the system call your action method when the button is tapped.

Note that because you're not using Interface Builder, you don't need to define your action method with the `IBAction` attribute; you just define the action like any other method.

Your `init?(coder:)` initializer should look like this:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let button = UIButton(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
    button.backgroundColor = UIColor.redColor()
    button.addTarget(self, action: #selector(RatingControl.ratingButtonTapped(_:)),
  forControlEvents: .TouchDown)
    addSubview(button)
}
```

Checkpoint: Run your app. When you click the red square, you should see the "Button pressed" message in the console.



Now it's time to think about what pieces of information the `RatingControl` class needs to have in order to represent a rating. You'll need to keep track of a rating value—0, 1, 2, 3, 4, or 5—as well as the buttons that a user taps to set that rating. You can represent the rating value with an `Int`, and the buttons as an array of `UIButton` objects.

To add rating properties

1. In `RatingControl.swift`, find the class declaration line:

```
class RatingControl: UIView {
```

2. Below this line, add the following code:

```
// MARK: Properties

var rating = 0
var ratingButtons = [UIButton]()
```

Right now, you have one button in the view, but you need five total. To create a total of five buttons, use a `for-in` loop. A `for-in` loop iterates over a sequence, such as ranges of numbers, to execute a set of code multiple times. Instead of creating one button, the loop will create five.

To create a total of five buttons

1. In `RatingControl.swift`, find the `init?(coder:)` initializer:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let button = UIButton(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
    button.backgroundColor = UIColor.redColor()
    button.addTarget(self, action:
  #selector(RatingControl.ratingButtonTapped(_:)), forControlEvents:
  .TouchDown)
    addSubview(button)
}
```

2. Add a `for-in` loop around the last four lines, like this:

```
for _ in 0..<5 {
    let button = UIButton(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
    button.backgroundColor = UIColor.redColor()
    button.addTarget(self, action:
  #selector(RatingControl.ratingButtonTapped(_:)), forControlEvents:
  .TouchDown)
    addSubview(button)
}
```

   You can make sure the lines in the `for-in` loop are indented properly by selecting all of them and pressing Control-I.

   The half-open range operator (`..<`) doesn't include the upper number, so this range goes from `0` to `4` for a total of five loop iterations, drawing five buttons instead of just one. The underscore (_) represents a wildcard, which you can use when you don't need to know which iteration of the loop is currently executing.

3. Above the `addSubview(button)` line, add this:

```
ratingButtons += [button]
```

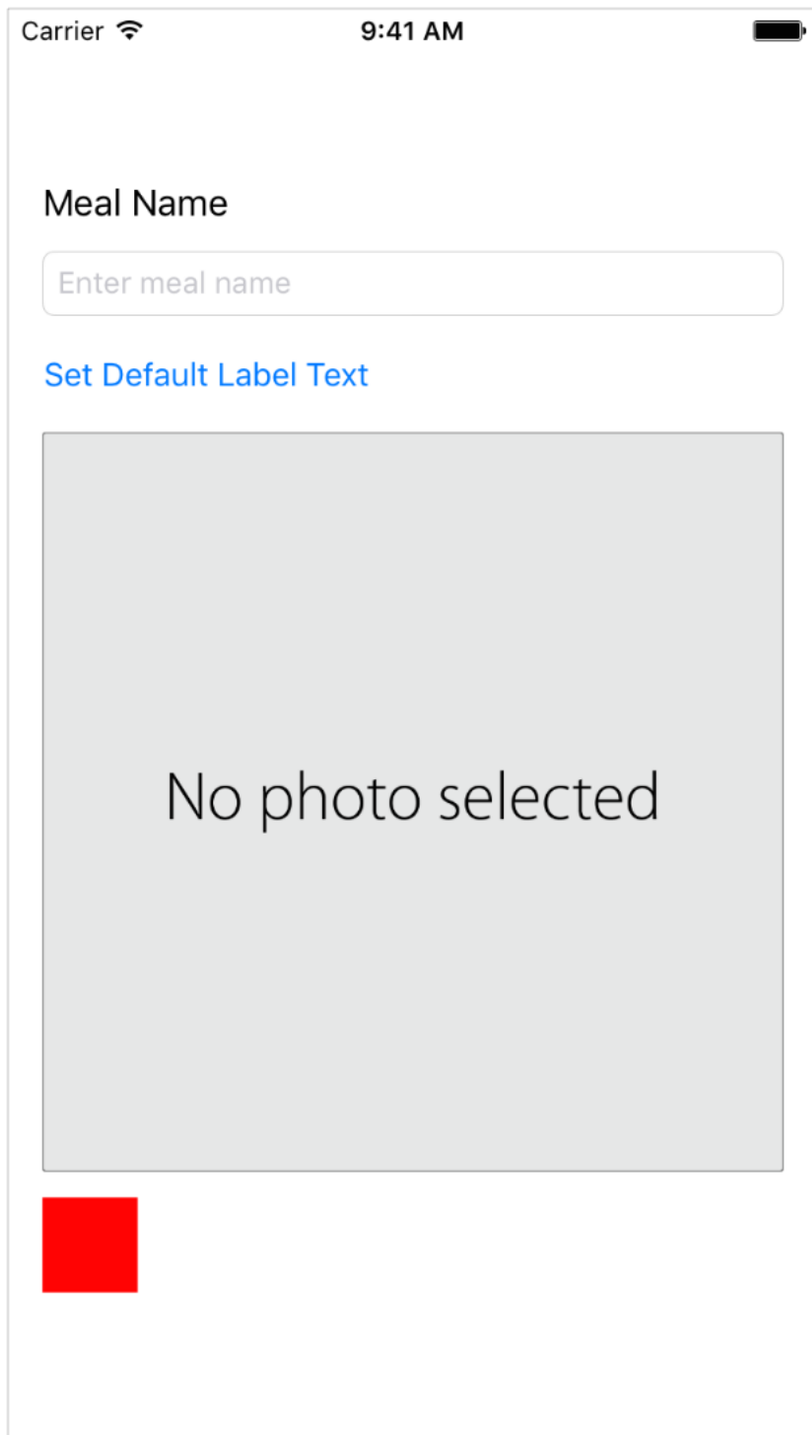
   As you create each button, you add it to the `ratingButtons` array to keep track of it.

Your `init?(coder:)` initializer should look like this:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    for _ in 0..<5 {
```

```
        let button = UIButton(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
        button.backgroundColor = UIColor.redColor()
        button.addTarget(self, action:
  #selector(RatingControl.ratingButtonTapped(_:)), forControlEvents: .TouchDown)
        ratingButtons += [button]
        addSubview(button)
    }
}
```

Checkpoint: Run your app. You'll notice it looks like there's only one button. That's because the `for-in` loop just stacked the buttons on top of each other. You'll need to adjust where the buttons lay out in the view.



This type of layout code belongs in a method called `layoutSubviews`, a method defined on the `UIView` class. The `layoutSubviews` method gets called at the appropriate time by the system and gives `UIView` subclasses a chance to perform a precise layout of their subviews. You'll need to override this method to place the buttons appropriately.

To lay out the buttons

1. In `RatingControl.swift`, under the `init?(coder:)` initializer in the `// MARK: Initialization` section, add the following method:

```
override func layoutSubviews() {
}
```

   Remember that you can use code completion to insert this method skeleton quickly.

2. In the method, add this code:

```
var buttonFrame = CGRect(x: 0, y: 0, width: 44, height: 44)

// Offset each button's origin by the length of the button plus spacing.
for (index, button) in ratingButtons.enumerate() {
    buttonFrame.origin.x = CGFloat(index * (44 + 5))
    button.frame = buttonFrame
```

```
}
```

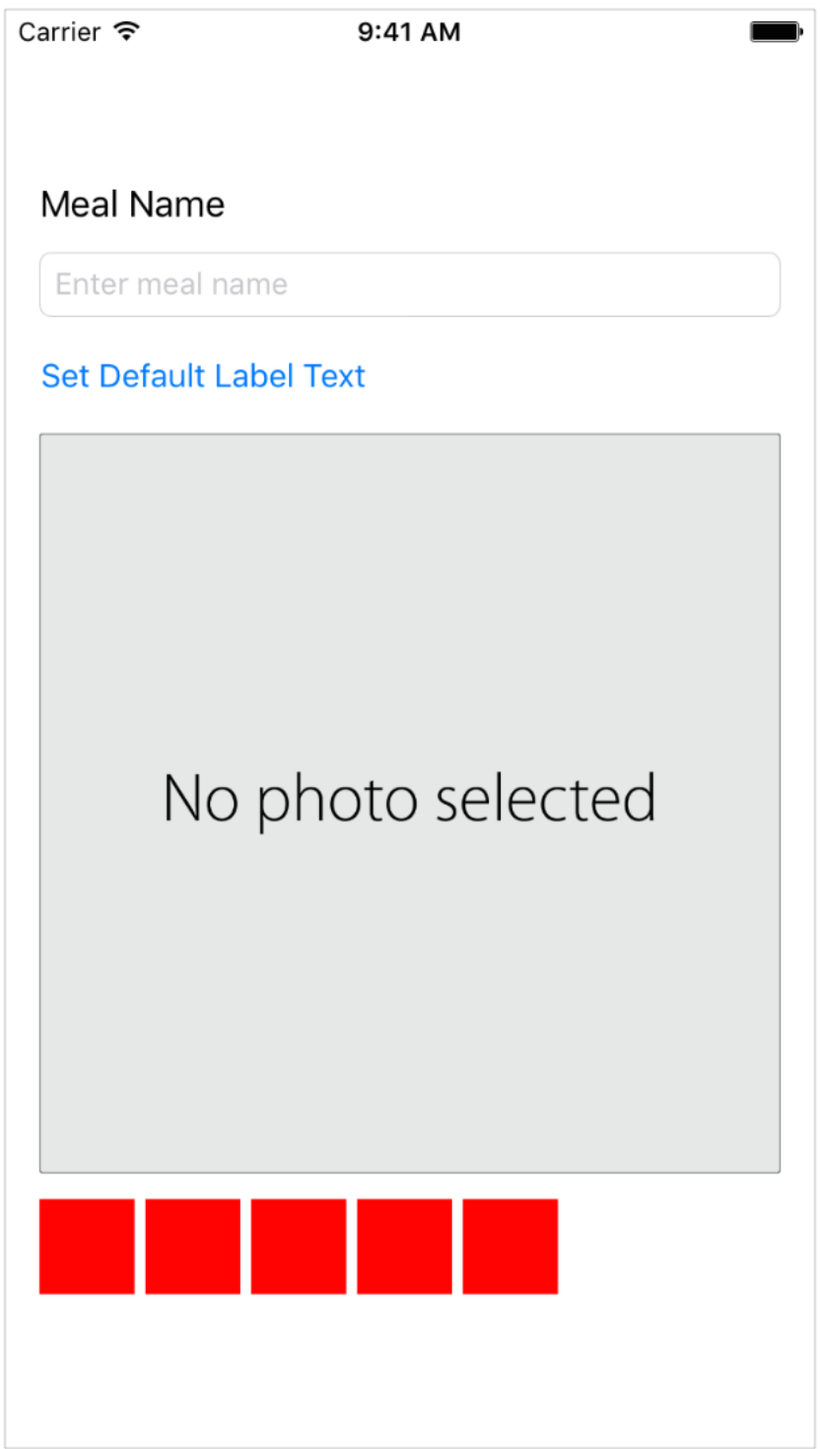
This code creates a frame, and uses a `for-in` loop to iterate over all of the buttons to set their frames.

The `enumerate()` method returns a collection that contains elements in the `ratingButtons` array paired with their indexes. This is a collection of tuples—groupings of values—and in this case, each tuple contains an index and a button. For each tuple in the collection, the `for-in` loop binds the values of the index and button in that tuple to local variables, `index` and `button`. You use the `index` variable to compute a new location for the button frame and set it on the `button` variable. The frame locations are set equal to a standard button size of `44` points and `5` points of padding, multiplied by `index`.

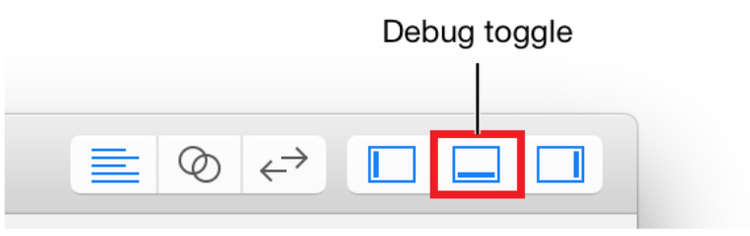
Your `layoutSubviews()` method should look like this:

```
override func layoutSubviews() {
    var buttonFrame = CGRect(x: 0, y: 0, width: 44, height: 44)

    // Offset each button's origin by the length of the button plus spacing.
    for (index, button) in ratingButtons.enumerate() {
        buttonFrame.origin.x = CGFloat(index * (44 + 5))
        button.frame = buttonFrame
    }
}
```

Checkpoint: Run your app. Now, the buttons should be side by side. Note, clicking any of the buttons at this point should still call `ratingButtonTapped(_:)` and log the message to the console.



To collapse the console, use the Debug area toggle.



Add Properties for Spacing and Number of Stars

Notice that you've been using the value `5` in your code, both for the number of stars and for the amount of space between stars. It's generally bad practice to keep hardcoded values scattered throughout your code. If you wanted to change the amount of space between stars, you'd have to change `5` wherever it was used to indicate space. The fact that the `5` values can mean two different things only further complicates the task.

Instead, declare two variables, one for the amount of space between stars and one for the number of stars. Then, if you need to make a change, you'll only need to change the value in one place.

To add properties

1. In `RatingControl.swift`, find the `// MARK: Properties` section.

   ```
   // MARK: Properties

   var rating = 0
   var ratingButtons = [UIButton]()
   ```

   You can jump to it quickly using the functions menu, which appears if you click the name of the file at the top of the editor area.

2. Below the existing properties, add the following code:

   ```
   let spacing = 5
   ```

   You add this constant property to define the spacing between buttons.

3. In `layoutSubviews`, replace the literal value you used for spacing with the `spacing` property:

   ```
   buttonFrame.origin.x = CGFloat(index * (44 + spacing))
   ```

4. Below the `spacing` property, add another property:

   ```
   let starCount = 5
   ```

   You can use this constant property to define the number of stars the control displays.

5. In `init?(coder:)`, replace the literal value you used for number of stars with the `starCount` property:

   ```
   for _ in 0..<starCount {
   ```

Checkpoint: Run your app. Everything should work and look exactly as before.

Declare a Constant for the Button Size

You have also been using the value `44` in your code to represent the button's size. Again, you want to get rid of hardcoded values wherever possible. This time, make your buttons adjust to the size of their container view (the view you added to your storyboard) by retrieving the container view's height. Use a local constant to store the container view's height, so you only need to access it once inside each method.

To declare a constant for the size of the buttons

1. In the `layoutSubviews()` method, add this code before the first line of the implementation:

   ```
   // Set the button's width and height to a square the size of the frame's
     height.
   let buttonSize = Int(frame.size.height)
   ```

   This makes the layout much more flexible.

2. Change the rest of the method to use the `buttonSize` constant instead of `44`:

```
var buttonFrame = CGRect(x: 0, y: 0, width: buttonSize, height: buttonSize)

// Offset each button's origin by the length of the button plus spacing.
for (index, button) in ratingButtons.enumerate() {
    buttonFrame.origin.x = CGFloat(index * (buttonSize + 5))
    button.frame = buttonFrame
}
```

3. As you did when you first added the rating control, you need to update the control's intrinsic content size so that the stack view can layout your control correctly. This time, you want the `intrinsicContentSize()` method to calculate the control's size accounting for each of the stars and the spaces between them (one less space than stars, assuming you have at least one star). Use code like this:

```
let buttonSize = Int(frame.size.height)
let width = (buttonSize * stars) + (spacing * (stars - 1))

return CGSize(width: width, height: buttonSize)
```

4. In the `init?(coder:)` initializer, change the first line of the `for-in` loop to this:

```
let button = UIButton()
```

Because you set button frames in `layoutSubviews()`, you no longer need to set them when you create the buttons.

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

```
override func layoutSubviews() {
    // Set the button's width and height to a square the size of the frame's height.
    let buttonSize = Int(frame.size.height)
    var buttonFrame = CGRect(x: 0, y: 0, width: buttonSize, height: buttonSize)

    // Offset each button's origin by the length of the button plus some spacing.
    for (index, button) in ratingButtons.enumerate() {
        buttonFrame.origin.x = CGFloat(index * (buttonSize + 5))
        button.frame = buttonFrame
    }
}
```

Your `intrinsicContentSize` method should look like this:

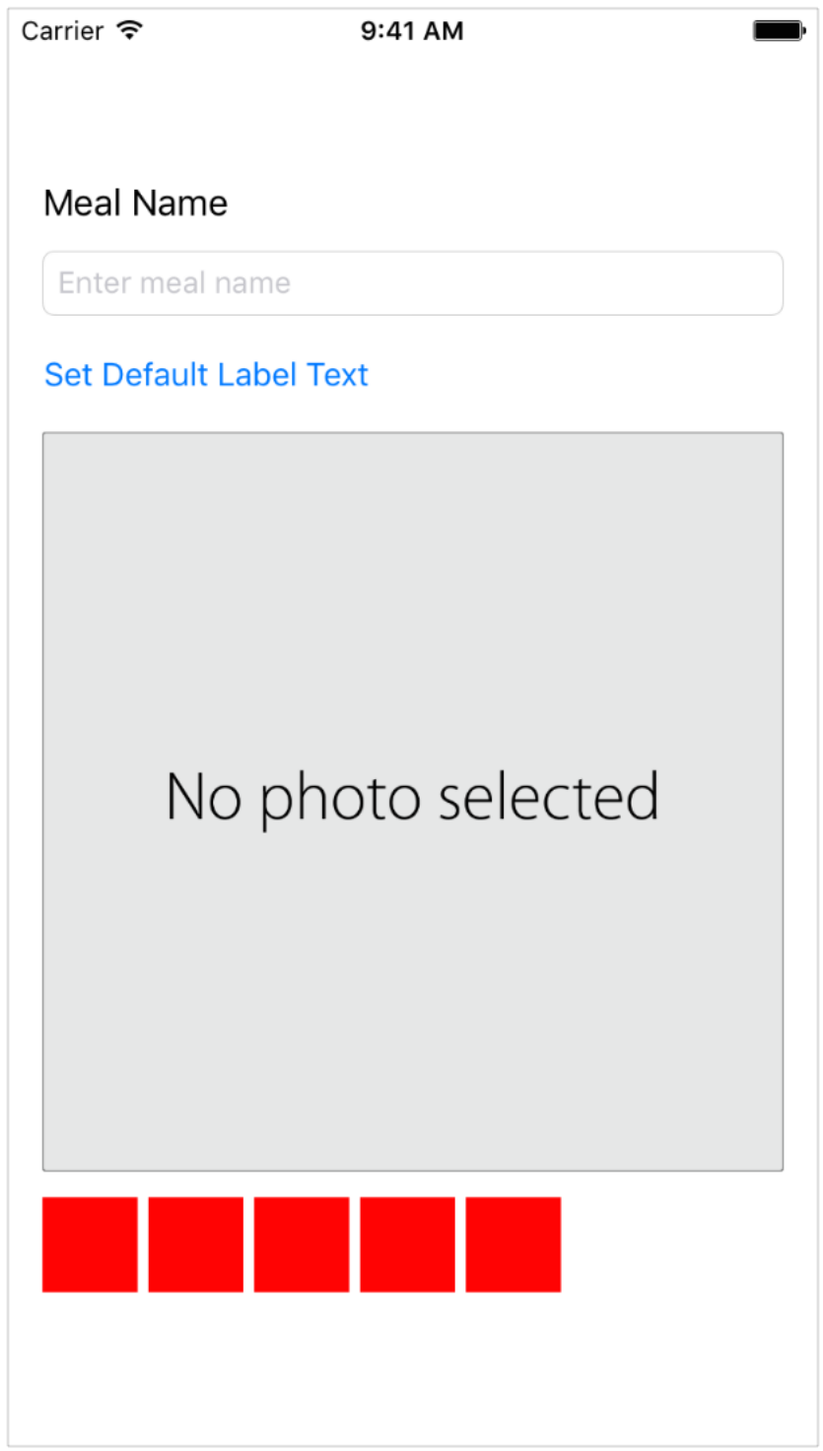
```
override func intrinsicContentSize() -> CGSize {
    let buttonSize = Int(frame.size.height)
    let width = (buttonSize * stars) + (spacing * (stars - 1))

    return CGSize(width: width, height: buttonSize)
}
```

And your `init?(coder:)` initializer should look like this:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    for _ in 0..<5 {
        let button = UIButton()
        button.backgroundColor = UIColor.redColor()
        button.addTarget(self, action: #selector(RatingControl.ratingButtonTapped(_:)), forControlEvents: .TouchDown)
```

```
        ratingButtons += [button]
        addSubview(button)
    }
}
```

Checkpoint: Run your app. Everything should work and look exactly as before. The buttons should be side-by-side. Clicking any of the buttons at this point should still call `ratingButtonTapped(_:)` and log the message to the console.



Add Star Images to the Buttons

Next, you'll add images of an empty and filled-in star to the buttons.

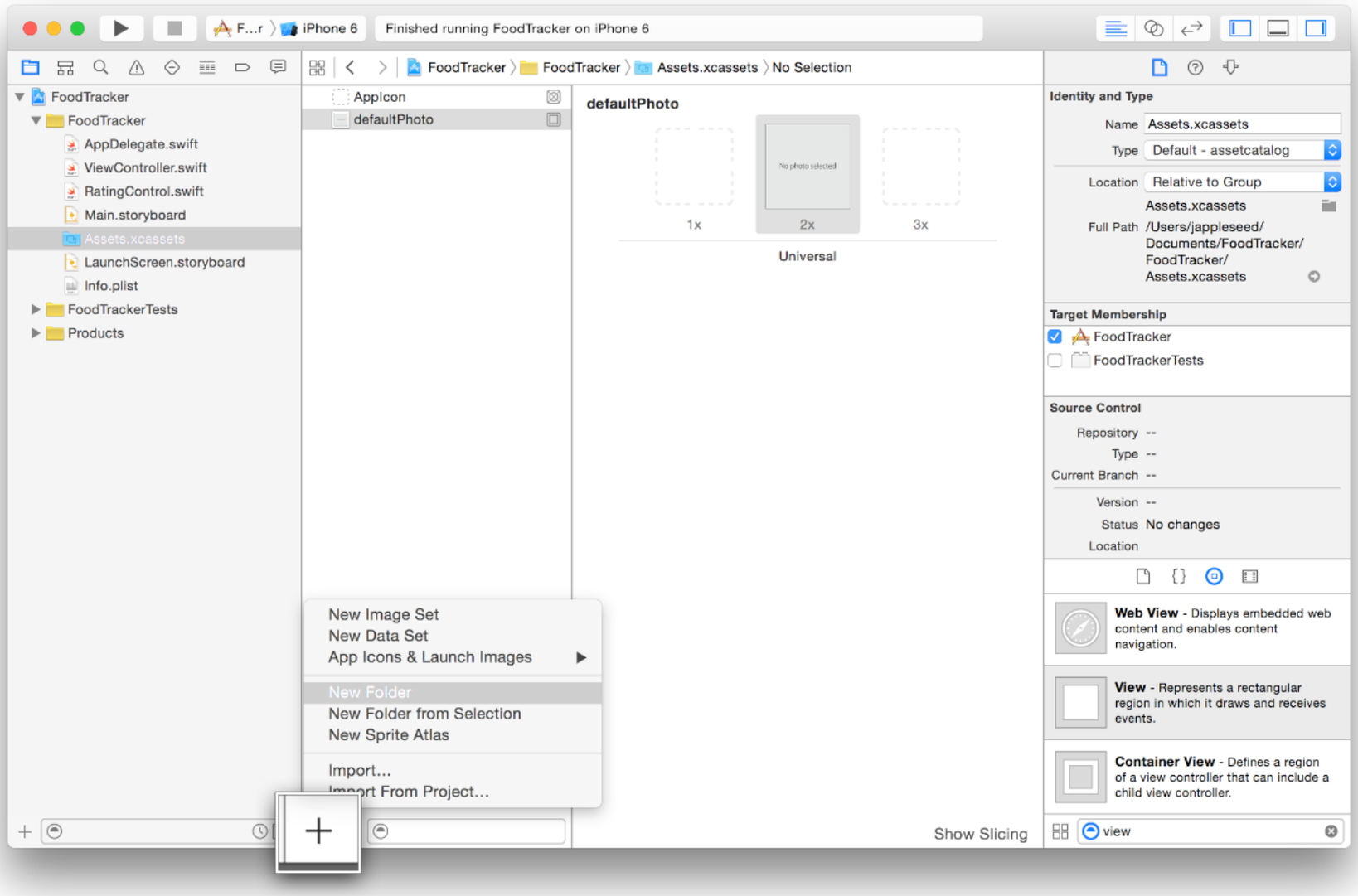
You can find the images shown above within the `Images/` folder of the downloadable file at the end of this lesson, or use your own images. (Just make sure the names of the images you use match the image names in the code later.)

To add images to your project

1. In the project navigator, select `Assets.xcassets` to view the asset catalog.

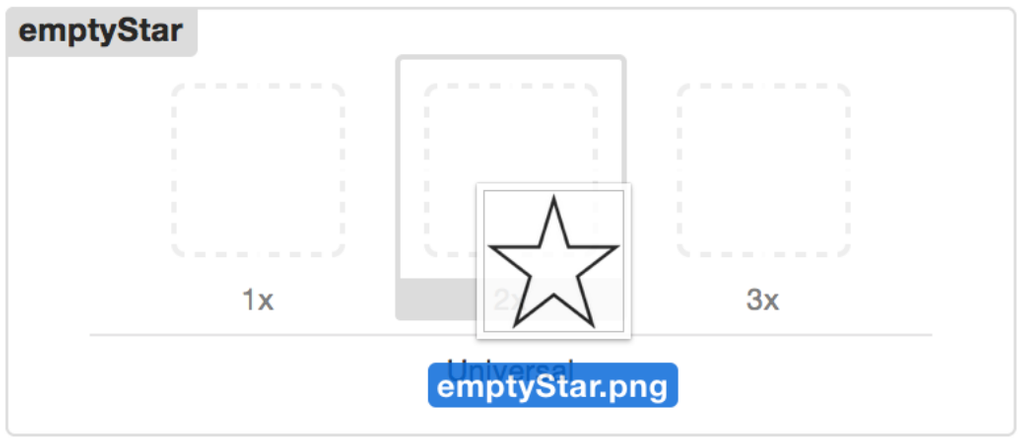
   Recall that the asset catalog is a place to store and organize your image assets for an app.

2. In the bottom left corner, click the plus (+) button and choose New Folder from the pop-up menu.

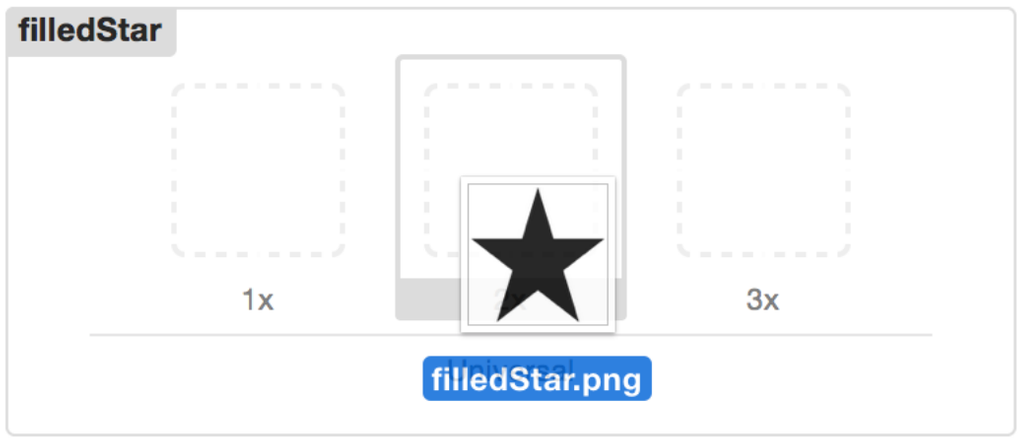


3. Double-click the folder name and rename it `Rating Images`.
4. With the folder selected, in the bottom left corner, click the plus (+) button and choose New Image Set from the pop-up menu.

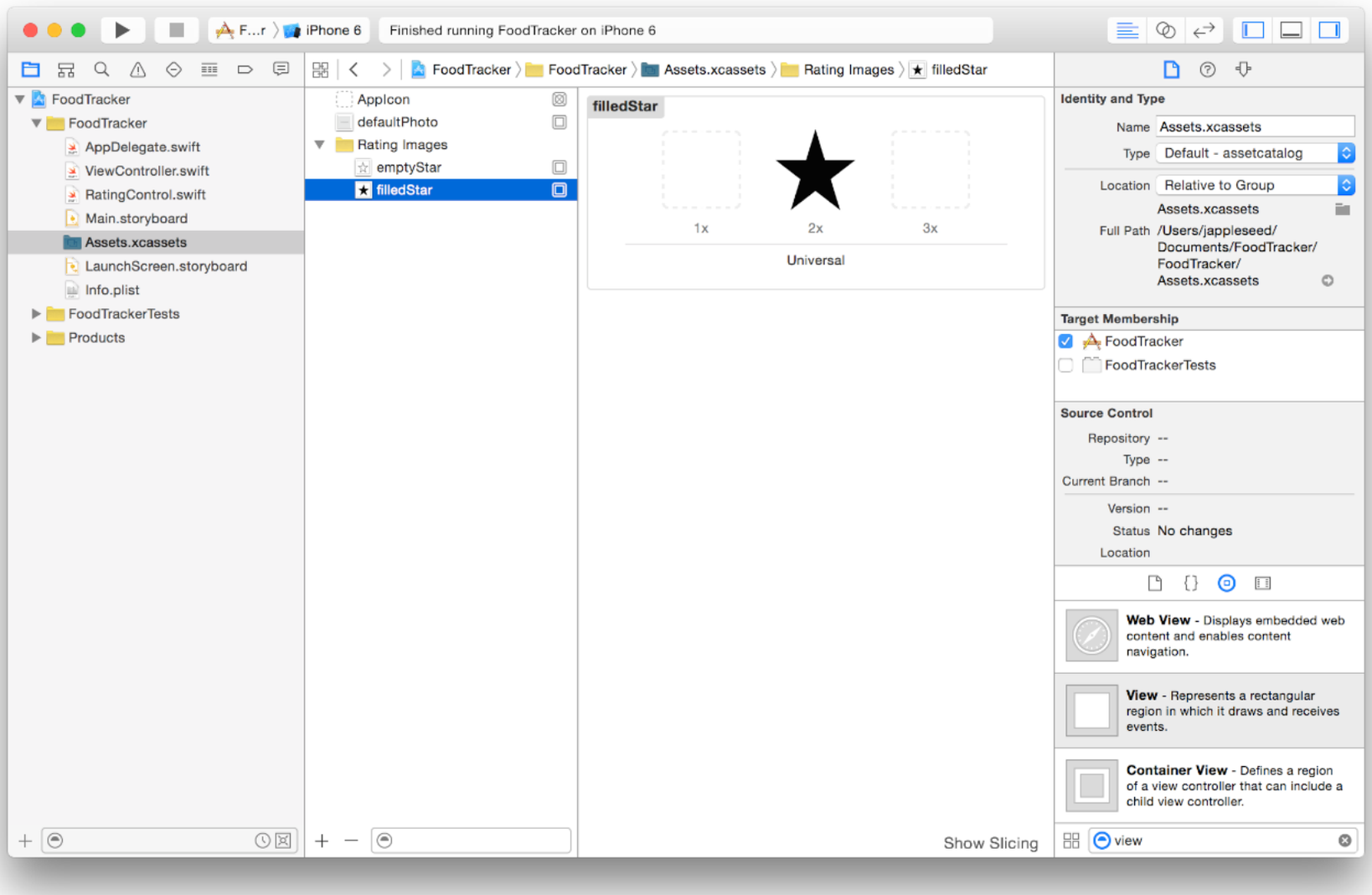
   An image set represents a single image asset, but can contain different versions of the image to display at different screen resolutions.
5. Double-click the image set name and rename it `emptyStar`.
6. On your computer, select the empty star image you want to add.
7. Drag and drop the image into the `2x` slot in the image set.



   `2x` is the display resolution for iPhone 6 Simulator that you're using in these lessons, so the image will look best at this resolution.
8. In the bottom left corner, click the plus (+) button and choose New Image Set from the pop-up menu.
9. Double-click the image set name and rename it `filledStar`.
10. On your computer, select the filled-in star image you want to add.
11. Drag and drop the image into the `2x` slot in the image set.



Your asset catalog should look something like this:



Next, write the code to set the appropriate image for a button at the right time.

To set star images for the buttons

1. Open `RatingControl.swift`.
2. In the `init?(coder:)` initializer, add these two lines of code before the `for-in` loop:

```
let filledStarImage = UIImage(named: "filledStar")
let emptyStarImage = UIImage(named: "emptyStar")
```

3. In the `for-in` loop, after the line where the button is initialized, add this code:

```
button.setImage(emptyStarImage, forState: .Normal)
button.setImage(filledStarImage, forState: .Selected)
button.setImage(filledStarImage, forState: [.Highlighted, .Selected])
```

   You're setting two different images for different states so you can see when the buttons have been selected. The empty star image appears when a button is unselected (`.Normal` state). The filled-in star image appears when the button is selected (`.Selected` state) and when the button is both selected and highlighted (`.Selected` and `.Highlighted` states), which occurs when a user is in the process of tapping the button.
4. Delete the line of code that sets the background color to red:

```
button.backgroundColor = UIColor.redColor()
```

   Because your buttons have images now, it's time to remove the background color.
5. Add this line of code:

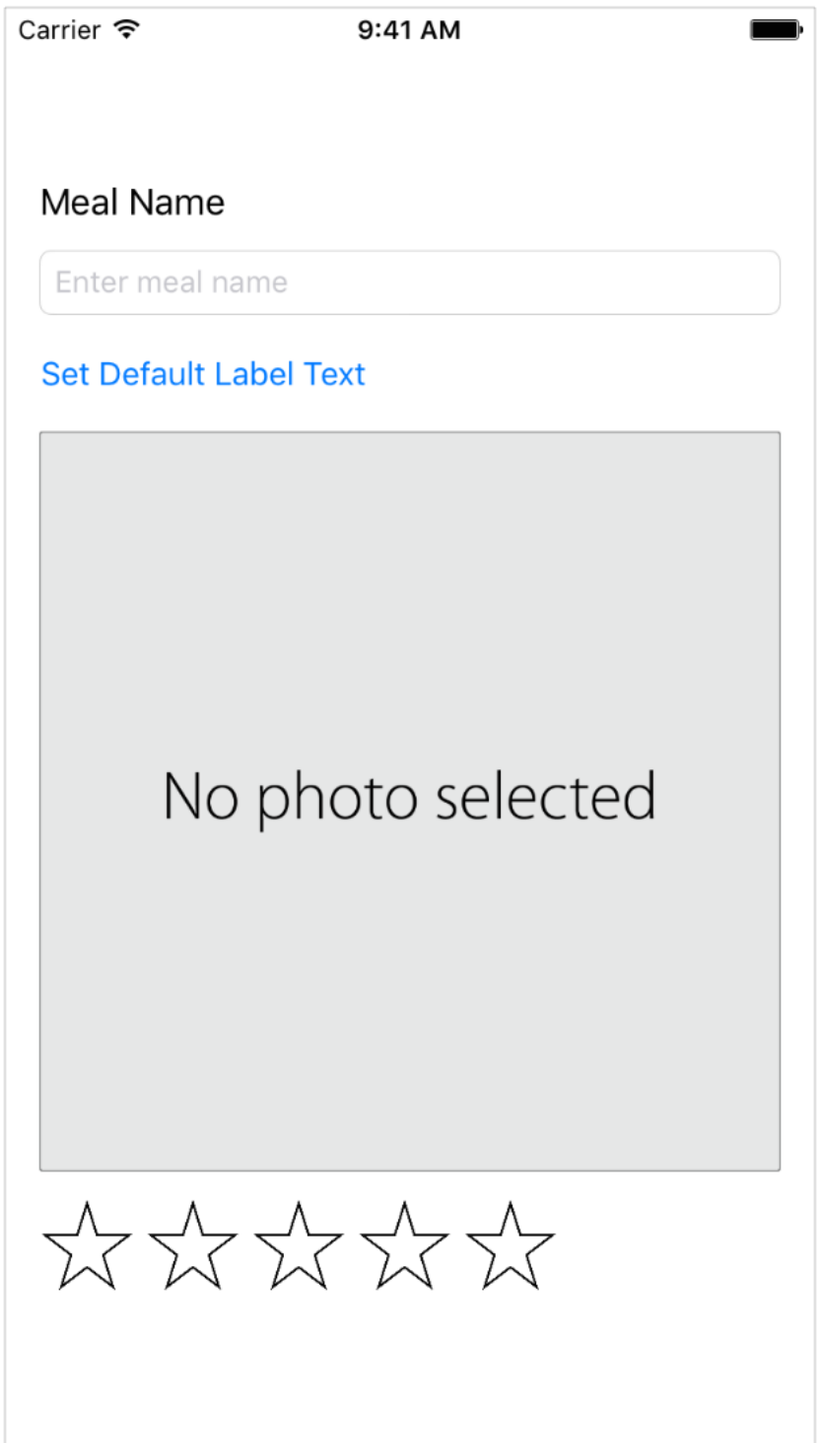
```
button.adjustsImageWhenHighlighted = false
```

This is to make sure that the image doesn't show an additional highlight during the state change.

Your `init?(coder:)` initializer should look like this:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let emptyStarImage = UIImage(named: "emptyStar")
    let filledStarImage = UIImage(named: "filledStar")

    for _ in 0..<5 {
        let button = UIButton()

        button.setImage(emptyStarImage, forState: .Normal)
        button.setImage(filledStarImage, forState: .Selected)
        button.setImage(filledStarImage, forState: [.Highlighted, .Selected])

        button.adjustsImageWhenHighlighted = false

        button.addTarget(self, action: #selector(RatingControl.ratingButtonTapped(_:)), forControlEvents: .TouchDown)
        ratingButtons += [button]
        addSubview(button)
    }
}
```

Checkpoint: Run your app. You should see stars instead of red buttons. Clicking any of the buttons at this point should still call `ratingButtonTapped(_:)` and log the message to the console, but your buttons don't change images yet. You'll fix that next.



Implement the Button Action

The user needs to be able to select a rating by tapping a star, so you'll replace the debugging implementation with a real implementation of the `ratingButtonTapped(_:)` method.

To implement the rating action

1. In `RatingControl.swift`, find the `ratingButtonTapped(_:)` method:

```
func ratingButtonTapped(button: UIButton) {
    print("Button pressed 👍")
}
```

2. Replace the `print` statement with this code:

```
rating = ratingButtons.indexOf(button)! + 1
```

   The `indexOf(_:)` method attempts to find the selected button in the array of buttons and to return the index at which it was found. This method returns an optional `Int` because the instance you're searching for might not exist in the collection you're searching. However, because the only buttons that trigger this action are the ones you created and added to the array yourself, you can be sure that searching for the button will return a valid index. In this case, you can use the force unwrap operator (`!`) to access the underlying index value. You add `1` to that index to get the corresponding rating. You need to add `1` because arrays are indexed starting with `0`.

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

```
func updateButtonSelectionStates() {
}
```

   This is a helper method that you'll use to update the selection state of the buttons.

4. In the `updateButtonSelectionStates()` method, add this `for-in` loop:

```
for (index, button) in ratingButtons.enumerate() {
    // If the index of a button is less than the rating, that button should be selected.
    button.selected = index < rating
}
```

   This code iterates through the button array to set the state of each button according to whether its index in the array is less than the rating. If it is, `index < rating` evaluates to `true`, which sets the button's state to selected and makes it display the filled-in star image. Otherwise, the button is unselected and shows the empty star image.

5. In the `ratingButtonTapped(_:)` method, add a call to `updateButtonSelectionStates()` as the last line of the implementation:

```
func ratingButtonTapped(button: UIButton) {
    rating = ratingButtons.indexOf(button)! + 1

    updateButtonSelectionStates()
}
```

6. In the `layoutSubviews()` method, add a call to `updateButtonSelectionStates()` as the last line of the implementation:

```
override func layoutSubviews() {
    // Set the button's width and height to a square the size of the frame's height.
    let buttonSize = Int(frame.size.height)
    var buttonFrame = CGRect(x: 0, y: 0, width: buttonSize, height: buttonSize)
```

```
    // Offset each button's origin by the length of the button plus some spacing.
    for (index, button) in ratingButtons.enumerate() {
        buttonFrame.origin.x = CGFloat(index * (buttonSize + 5))
        button.frame = buttonFrame
    }
    updateButtonSelectionStates()
}
```

It's important to update the button selection states when the view loads, not just when the rating changes.

7. In the `// MARK: Properties` section, find the `rating` property:

```
var rating = 0
```

8. Update the `rating` property to include this property observer:

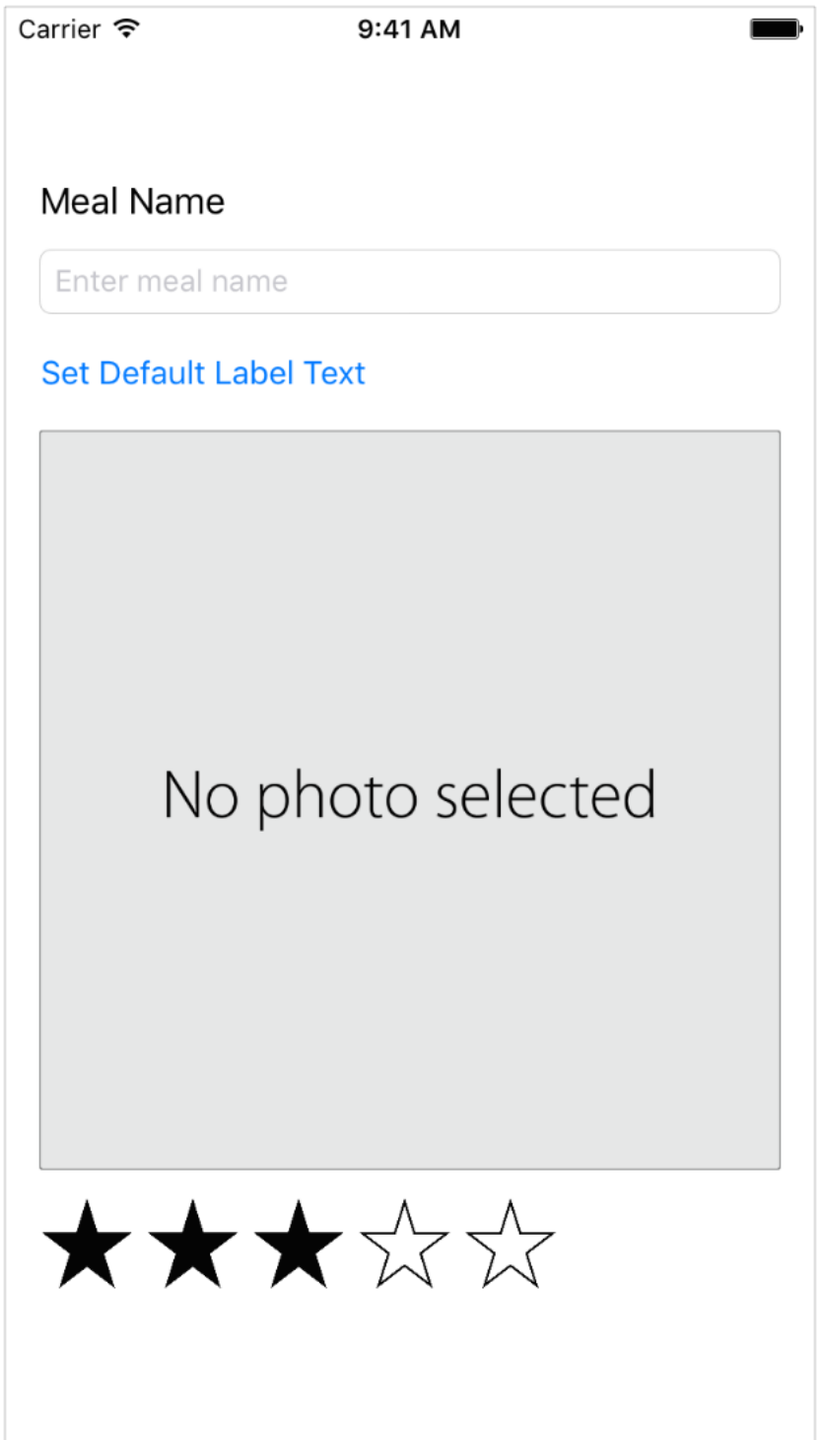
```
var rating = 0 {
didSet {
    setNeedsLayout()
}
}
```

A property observer observes and responds to changes in a property's value. Property observers are called every time a property's value is set, and can be used to perform work immediately before or after the value changes. Specifically, the `didSet` property observer is called immediately after the property's value is set. Here, you include a call to `setNeedsLayout()`, which will trigger a layout update every time the rating changes. This ensures that the UI is always showing an accurate representation of the `rating` property value.

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

```
func updateButtonSelectionStates() {
    for (index, button) in ratingButtons.enumerate() {
        // If the index of a button is less than the rating, that button shouldn't be selected.
        button.selected = index < rating
    }
}
```

Checkpoint: Run your app. You should see five stars and be able to click one to change the rating. Click the third star to change the rating to 3, for example.

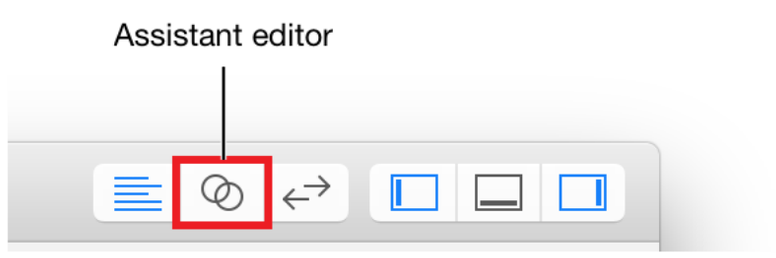


Connect the Rating Control to the View Controller

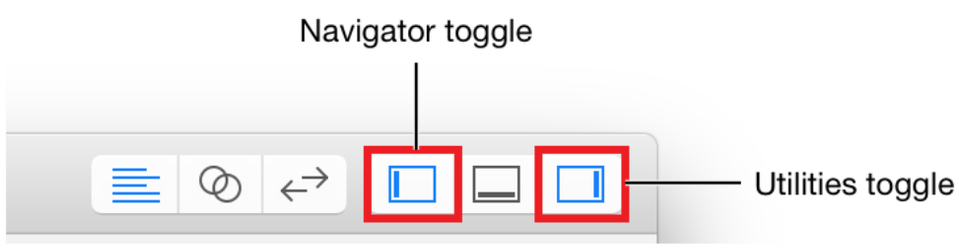
The last thing you need to do to set up the rating control is to give the `ViewController` class a reference to it.

To connect a rating control outlet to ViewController.swift

1. Open your storyboard.
2. Click the Assistant button in the Xcode toolbar 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.

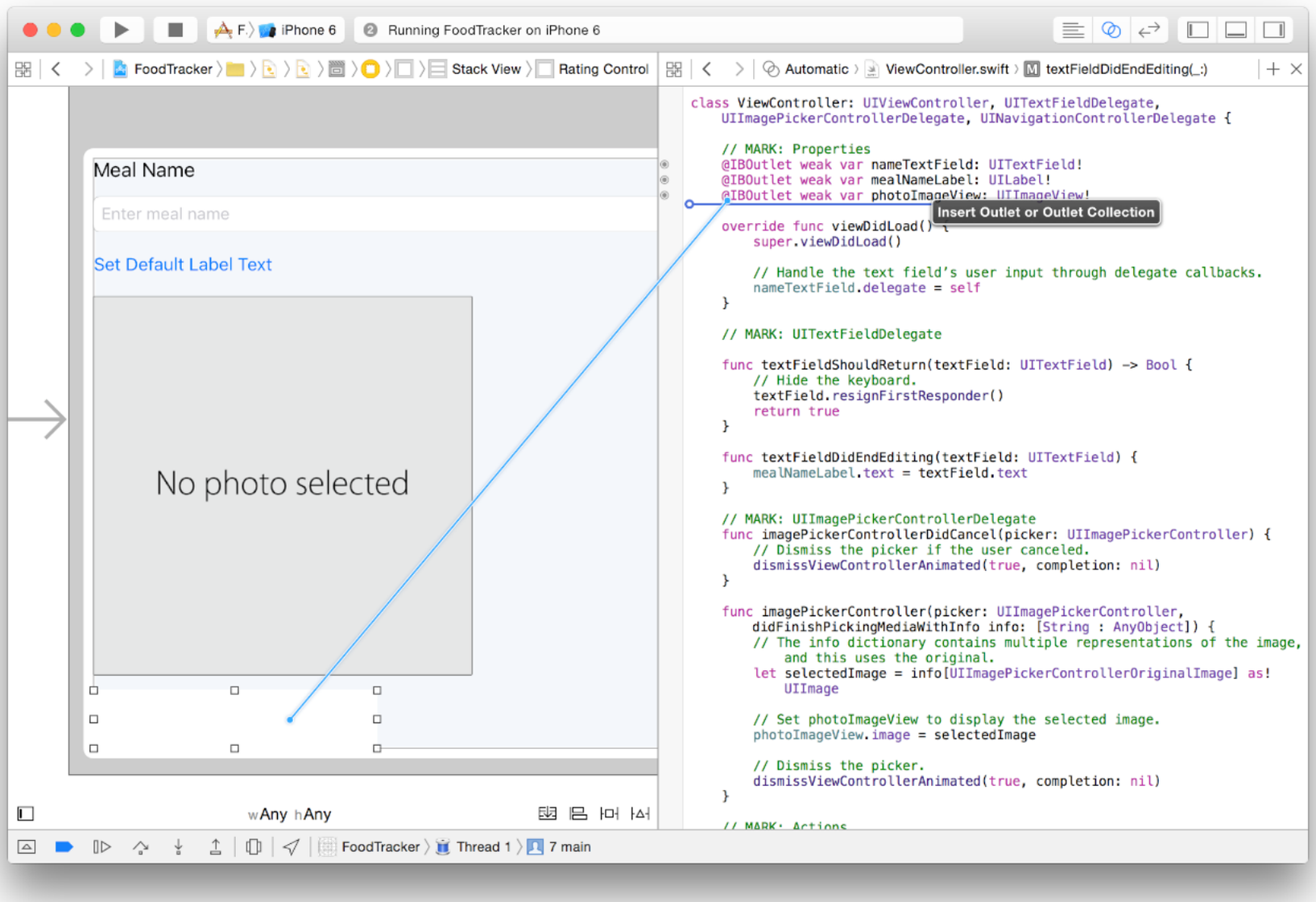


   You can also collapse the outline view.

4. Select the rating control.

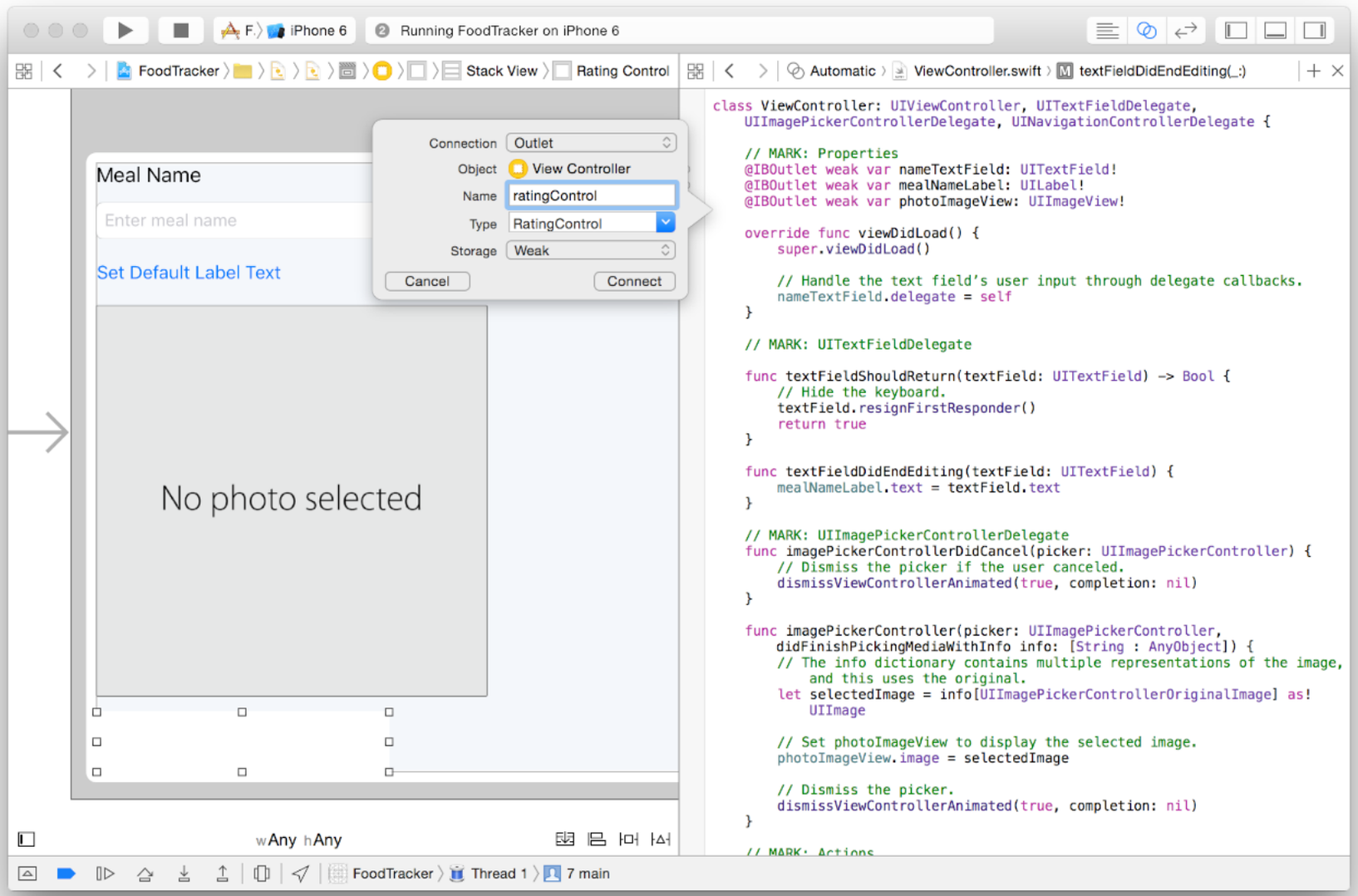
   `ViewController.swift` displays in the editor on the right. (If it doesn't, choose Automatic > ViewController.swift in the editor selector bar)

5. Control-drag from the rating control on your canvas to the code display in the editor on the right, stopping the drag at the line below the `photoImageView` property in `ViewController.swift`.



6. In the dialog that appears, for Name, type `ratingControl`.

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



7. Click Connect.

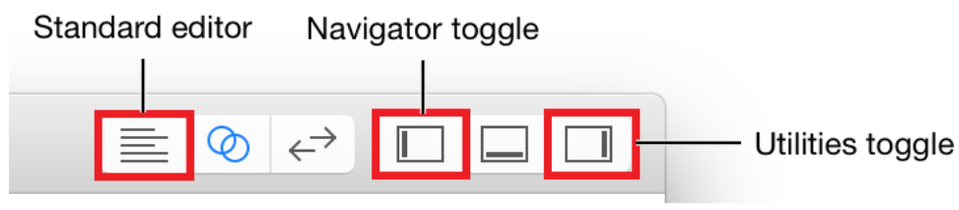
The `ViewController` class now has a reference to the rating control in the storyboard.

Clean Up the Project

You're close to finalizing the meal scene UI, but first you need to do some cleanup. Now that the FoodTracker app is implementing more advanced behavior and a different UI than in the previous lessons, you'll want to remove the pieces you don't need. You'll also center the elements in your stack view to balance the UI.

To clean up the UI

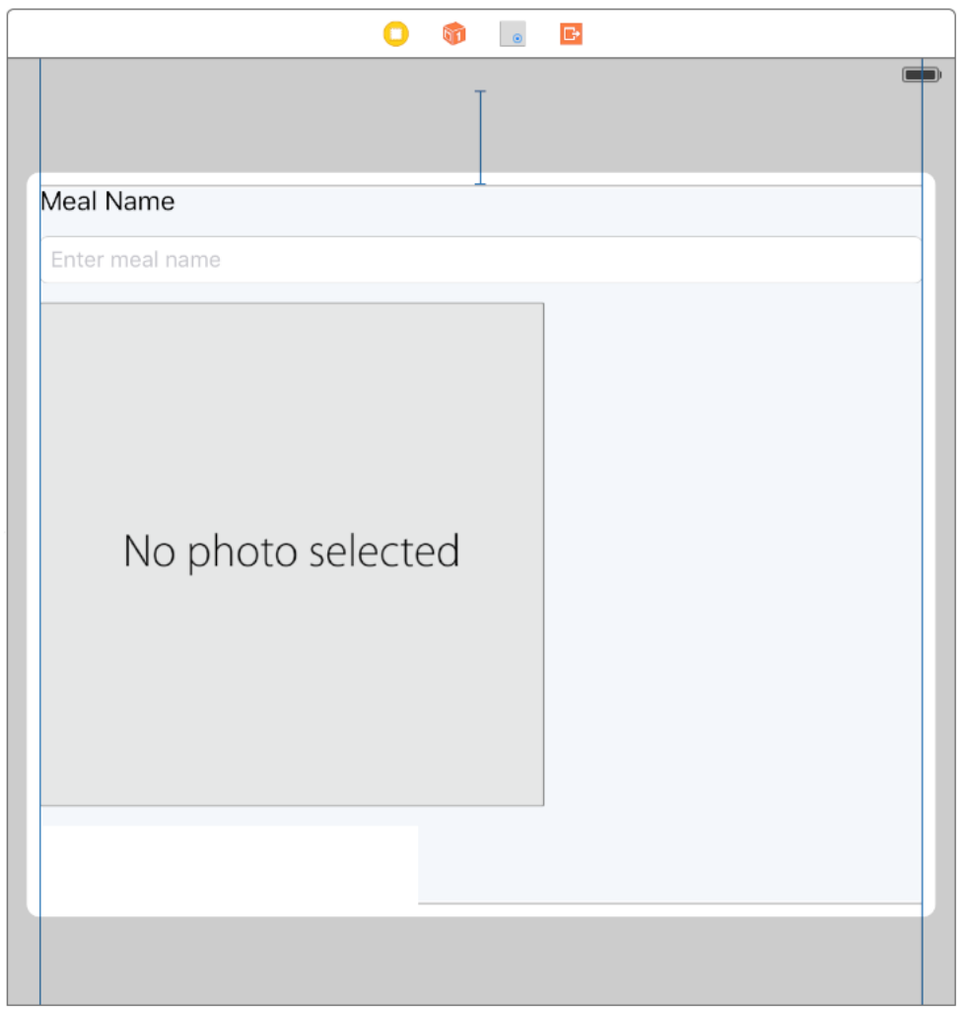
1. Return to the standard editor by clicking the Standard button.



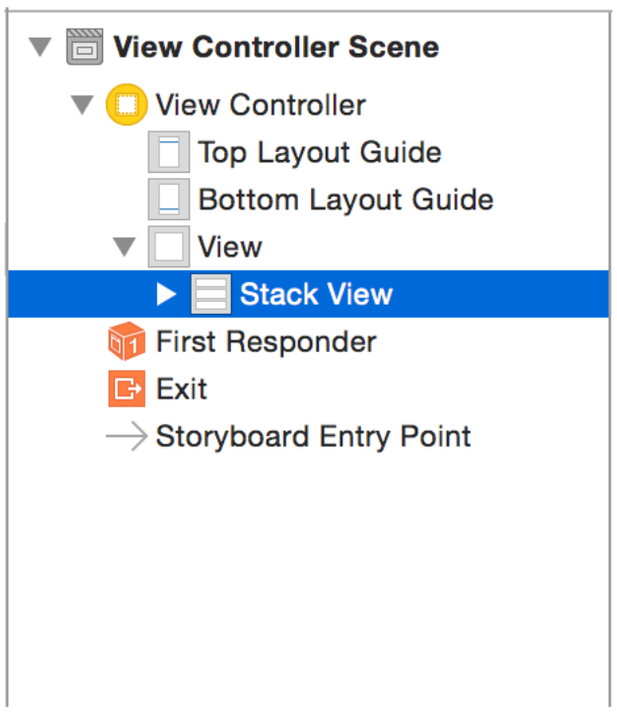
   Expand the project navigator and utility area by clicking the Navigator and Utilities buttons in the Xcode toolbar.

2. Open your storyboard.
3. Select the Set Default Label Text button, and press the Delete key to delete it.

   The stack view rearranges your UI elements to fill the gap that the button left.

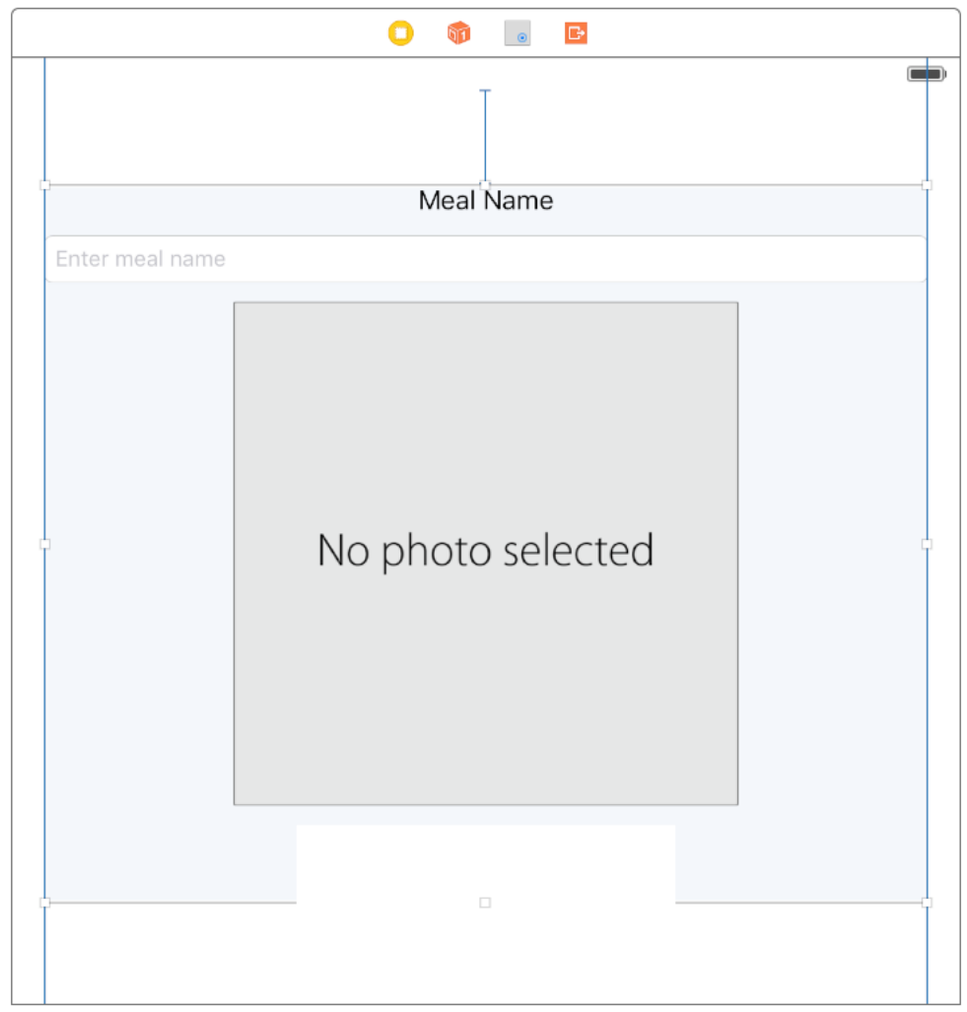


4. If necessary, open the outline view. Select the Stack View object.



5. Open the Attributes inspector.
6. In the Attributes inspector, find the Alignment field and select Center.

   The elements in the stack view center horizontally:



Now, remove the action method that corresponds with the button you deleted.

To clean up the code

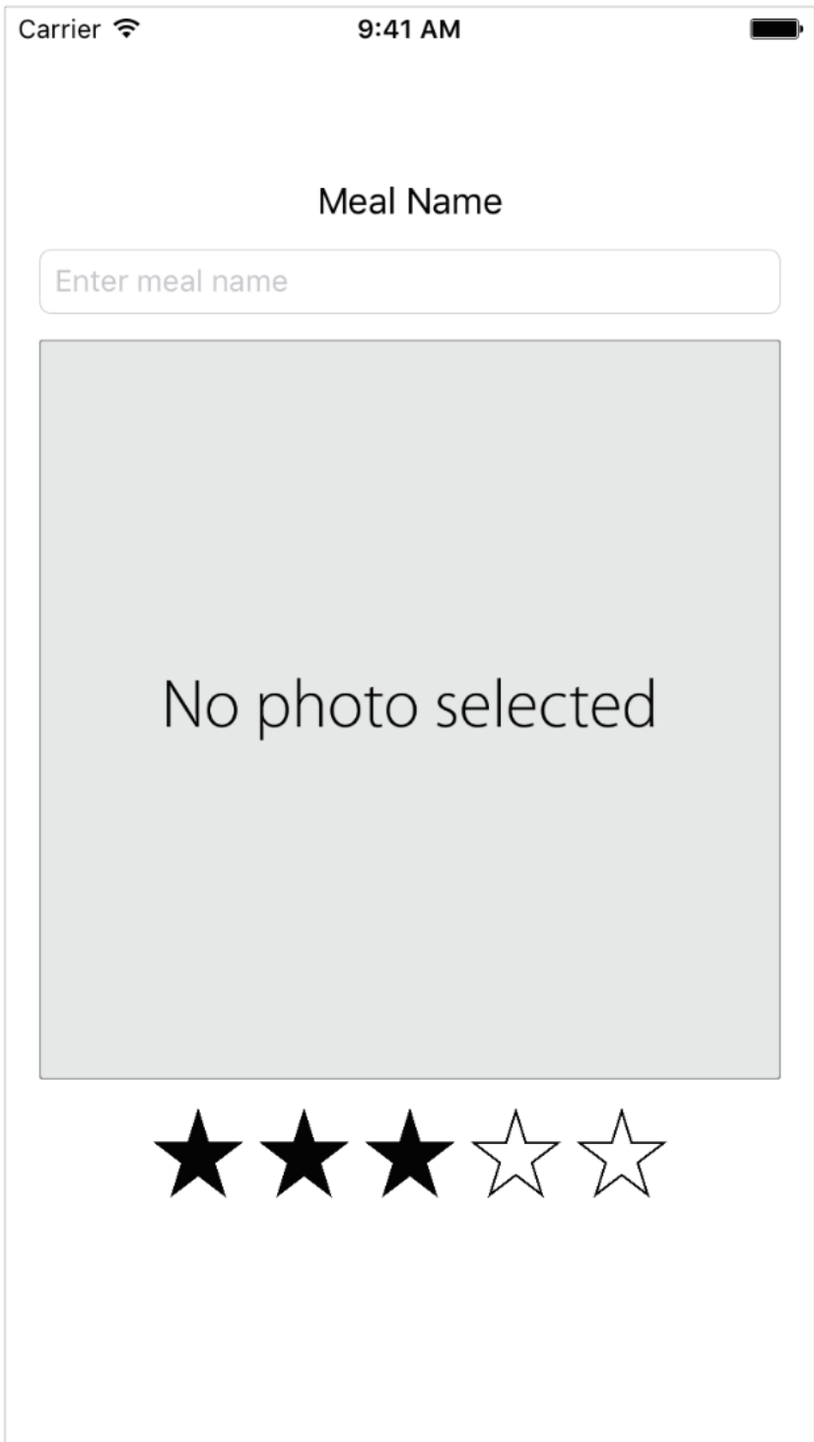
1. Open `ViewController.swift`.
2. In `ViewController.swift`, delete the `setDefaultLabelText(_:)` action method.

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

   That's all you need to delete for now. You'll make a change to the label outlet (`mealNameLabel`) in a later lesson.

Checkpoint: Run your app. Everything should work exactly as before, but the Set Default Label Text button is gone, and the elements are centered horizontally. The buttons should be side-by-side. Clicking any of the buttons at this point should still call `ratingButtonTapped(_:)` and change the button images appropriately.

> IMPORTANT
>
> If you're running into build issues, try pressing Command-Shift-K to clean your project.



NOTE

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

Download File