Chapter 16
Basic Animations, Visual Effects and Unwind Segues

> Animation can explain whatever the mind of man can conceive. This facility makes it the most versatile and explicit means of communication yet devised for quick mass appreciation.
>
> – Walk Disney

In iOS, creating sophisticated animations does not require you to write complex code. All you need to know is a single method in the `UIView` class:

```
UIView.animateWithDuration(1.0, animations)
```

There are several variations of the method that provide additional configuration and features. This is the basis of every view animation.

First things first, what's an animation? How is an animation created? Animation is a simulation of motion and shape change by rapidly displaying a series of static images (or frames). It is an illusion that an object is moving or changing in size. For instance, a growing circle animation is actually created by displaying a sequence of frames. It starts with a dot. The circle in each frame is a bit larger than the one before it. This creates an illusion that the dot grows bigger and bigger. Figure 16-1 illustrates the sequences of static images. I keep the example simple so the figure displays 5 frames. To achieve a smooth transition and animation, you'd need to develop several more frames.

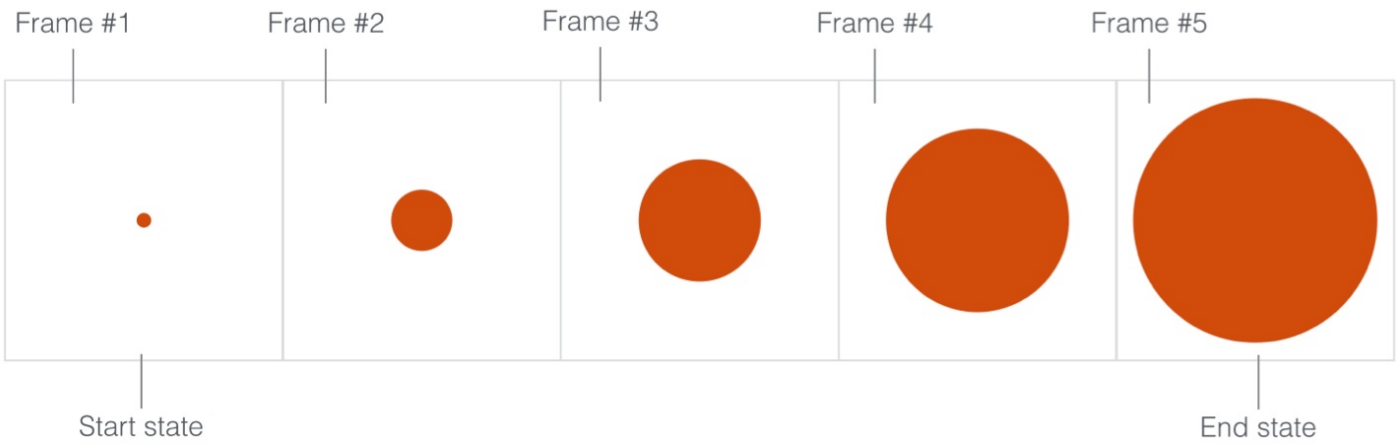

Figure 16-1. Sequence of frames for creating an animation

Now that you have a basic idea of how animation works, how would you create an animation in iOS? Consider our growing circle example. You know the animation starts with a dot (i.e. start state) and ends with a big red circle (i.e. end state). The challenge is to generate the frames between those states. You may need to think of an algorithm and write hundreds of lines of code to generate the series of frames in between. UIView animation helps you compute the frames between the start and end state resulting in a smooth animation. You simply specify the start state and tell UIView the end state by calling the `UIView.animateWithDuration` method.

The rest is handled by iOS. Sounds good, right?

There is no better way to understand the technique than by working on a real example. We will add some basic animations to our FoodPin app. Here is what we're going to do:

- Add a rating button in the detail view
- When a user taps the button, it brings up a review view controller, in which the buttons are animated, for the user to rate the restaurant.

Through building the review controller, I will show you how to create basic animations using `UIView`.

Adding a Rating button

Before creating the animated views, we'll add a rating button to the detail view controller. First, download this image pack (https://www.dropbox.com/s/jpv2daf5y0hb7xv/emoticons.zip?dl=0) and add the icons to `Assets.xcassets`.

```
Credit: The icons are made by Freepik from www.flaticon.com. It is licensed by
CC BY 3.0.
```

Go to `Main.storyboard` and drag a Button object from the Object library to the detail view controller. Place it at the top-right corner of the image view. In the Size inspector, change the button's width and height to `40` points. In the Attributes inspector, change the *image* option to `rating` and its type to `System`. Next, scroll down to set the *background* color to `red` and *tint* to `white`.

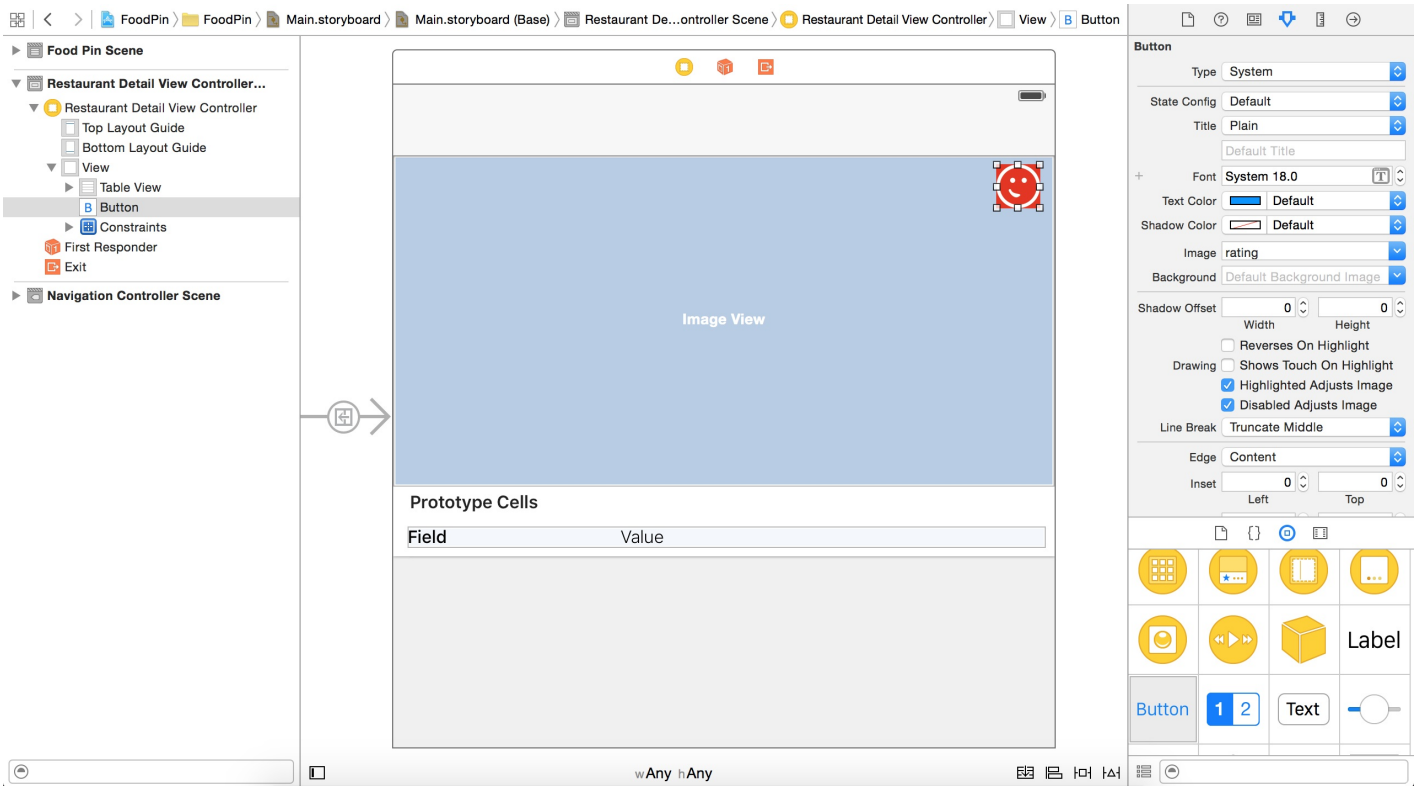

Figure 16-2. Adding a review button in the detail view

As usual, we need to add some layout constraints for the button. Select the button and click the Pin button in the layout bar. Refer to figure 16-3 to add four layout constraints (two spacing constraints for the top & right sides and two constraints to control the width & height of the button).

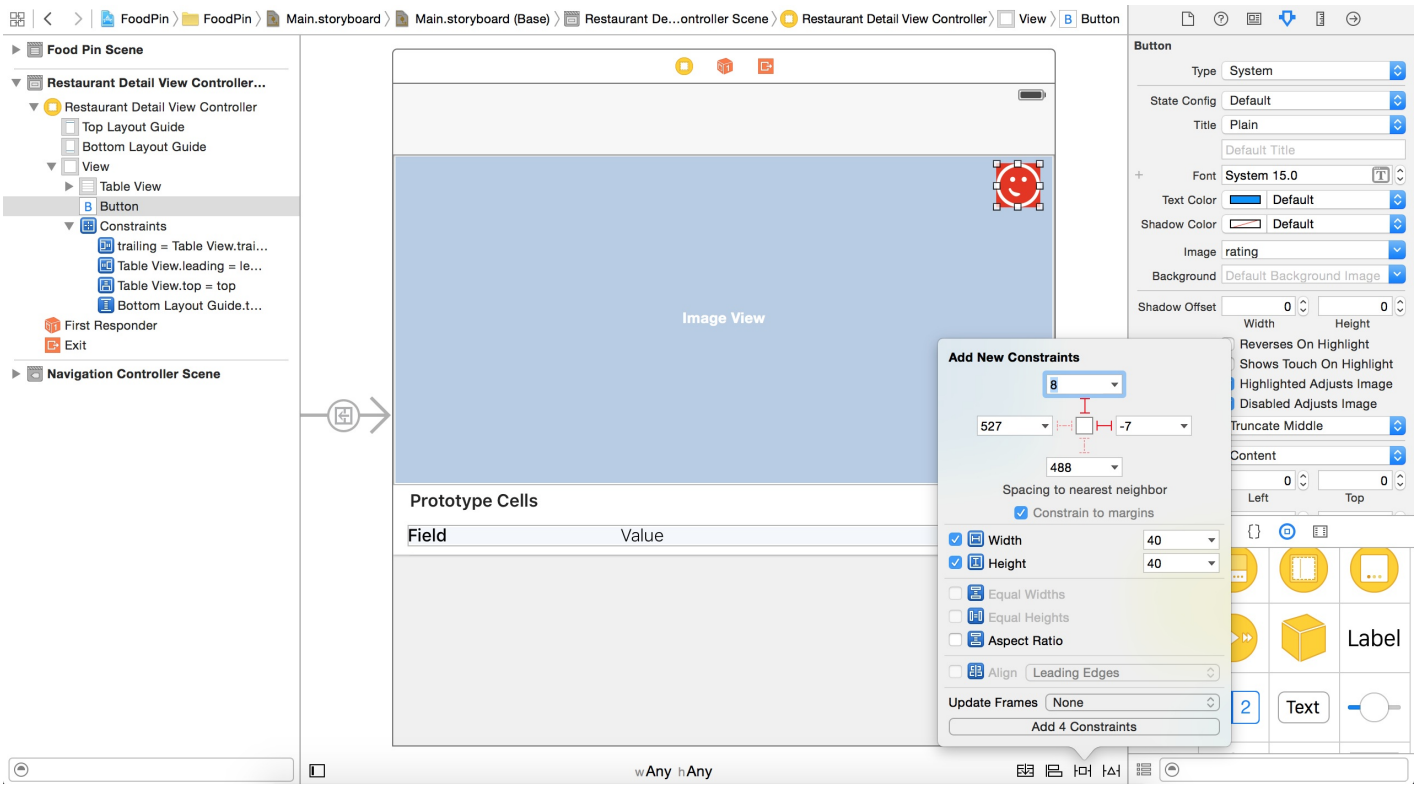

Figure 16-3. Adding layout constraints for the button

In order to create a circular button, go to the Identity inspector and add a runtime attribute for `layer.cornerRadius`. Set its value to 20, which is half of the button's width. Once done, run the app to have a quick test. Your app should look very similar to that in figure 16-4.

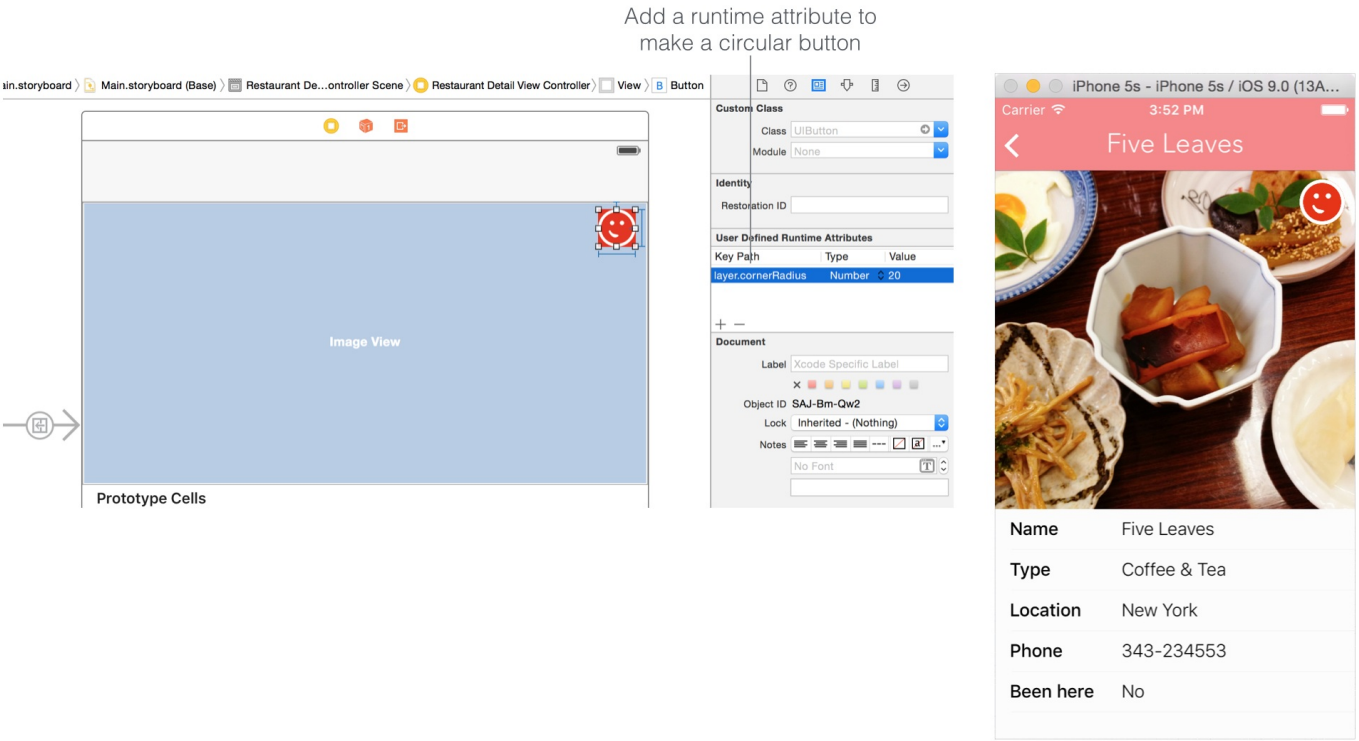

Figure 16-4. Adding a runtime attribute (left), The review button in the detail view (right)

Create a View Controller for Restaurant Review

When a user taps the *Review* button, we want to bring up a modal view for the user to give a rating for the restaurant. In Interface Builder, drag a new view controller from the Object library to the storyboard. Add an image view to the view and set the image to `barrafina`. Then select the "Resolve Auto Layout Issues" button in the layout bar and choose *Add Missing Constraints*. Xcode automatically adds the layout constraints for you.

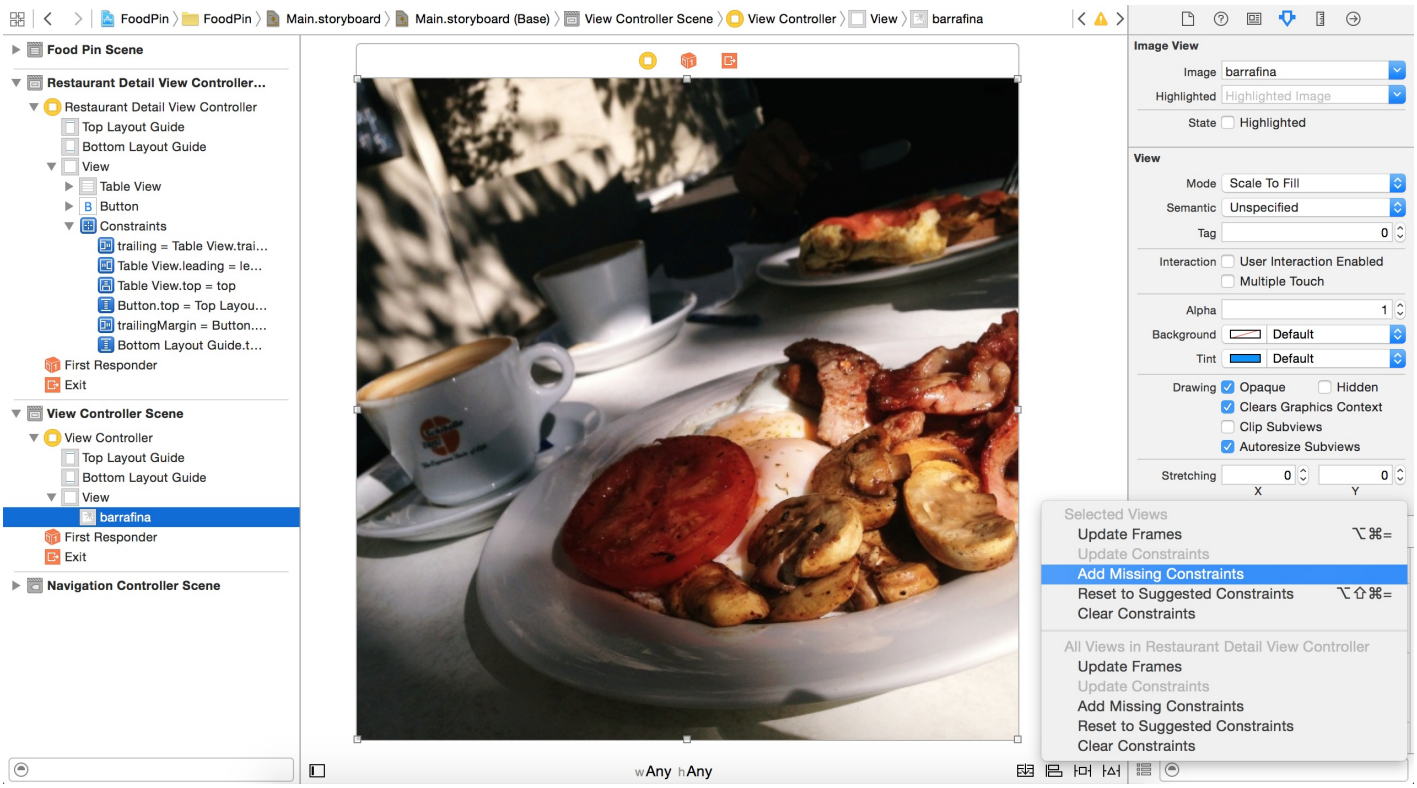

Figure 16-5. Adding missing constraints

```
Quick note: You may wonder why I do not show you this amazing option for
defining layout constraints. First, I want you to  understand auto layout
thoroughly, so you know how to define the constraints. Secondly, the "Add
Missing Constraints" option doesn't always work. It mostly works on simple
layouts. But for more complicated ones, you have to handle it on your own.
```

This image is used as the background image of the view. Later I'll show you how to apply a blurring effect to the image.

Now, drag a label to the image view. Change its title to "You've dined here. What did you think?" or whatever message you like. Change the font color to `white` and size to `30` points. Make sure you center the label horizontally to the image view.

Next, drag three buttons to the image view and change their size to 70x70 points. Then change the title and image for each of the image:

- Set the title of the first button to `blank` and the image to `dislike`.
- Set the title of the second button to `blank` and the image to `good`.

- Set the title of the third button to `blank` and the image to `great`.

Again, you can change the background to `red` and set the tint to `white`. Just make sure you change its type to `System` to effectuate the color change. To make the button circular, add a runtime attribute named `layer.cornerRadius` for each of the image and set its value to `35`. Your label and buttons should be similar to that in figure 16-6.

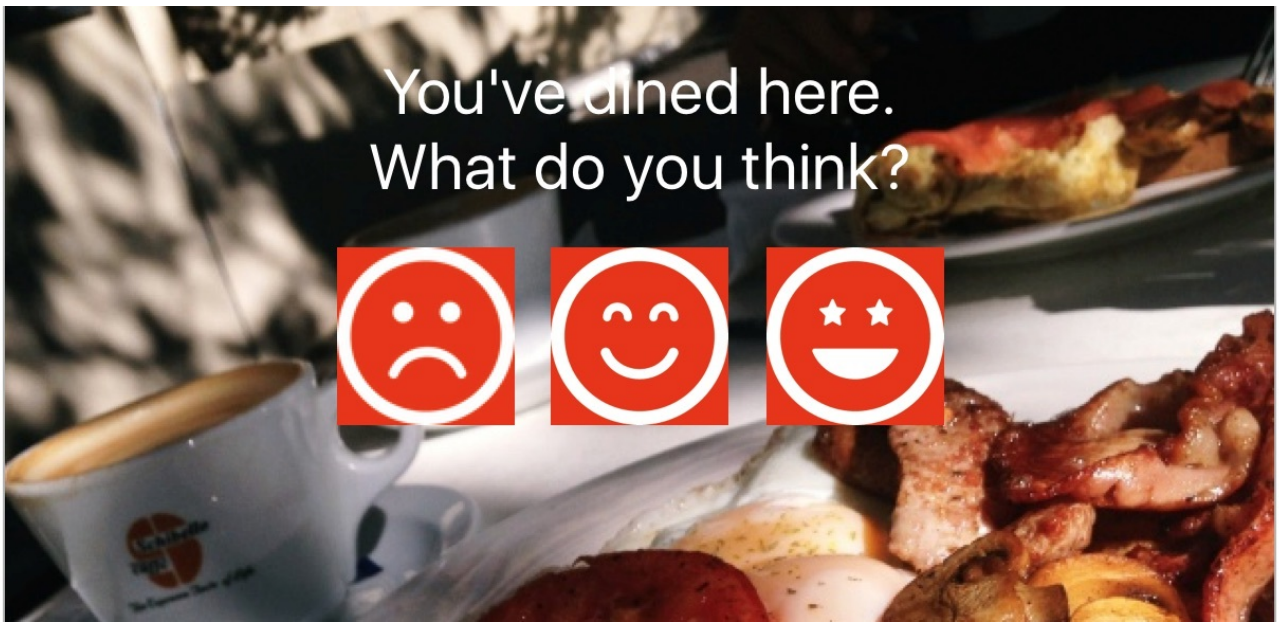

Figure 16-6. Label and rating buttons

Next, we will embed the buttons in a stack view, and then define the necessary layout constraints. Select the three buttons and click the *Stack* button in the layout bar to embed them in a horizontal stack view. The stack view tries its best to fit all the buttons, but it doesn't look good. Therefore, we have to define some extra constraints to control the size of the buttons. Control-drag diagonally on the `dislike` button, and add both *width* and *height* constraints.

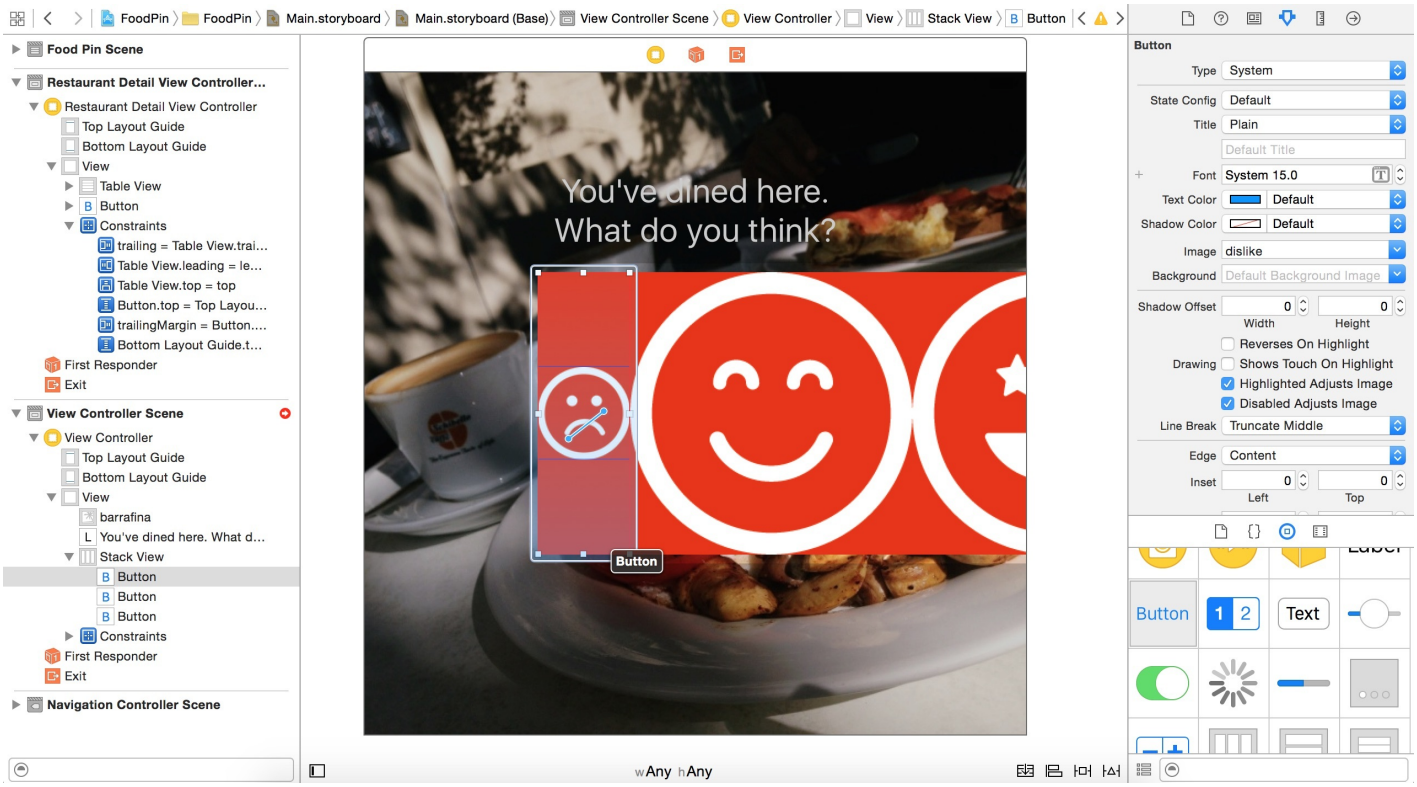

Figure 16-7. Control-drag diagonally to add the width and height constraints

Then select the width & height constraints and change the constant value to `70`.

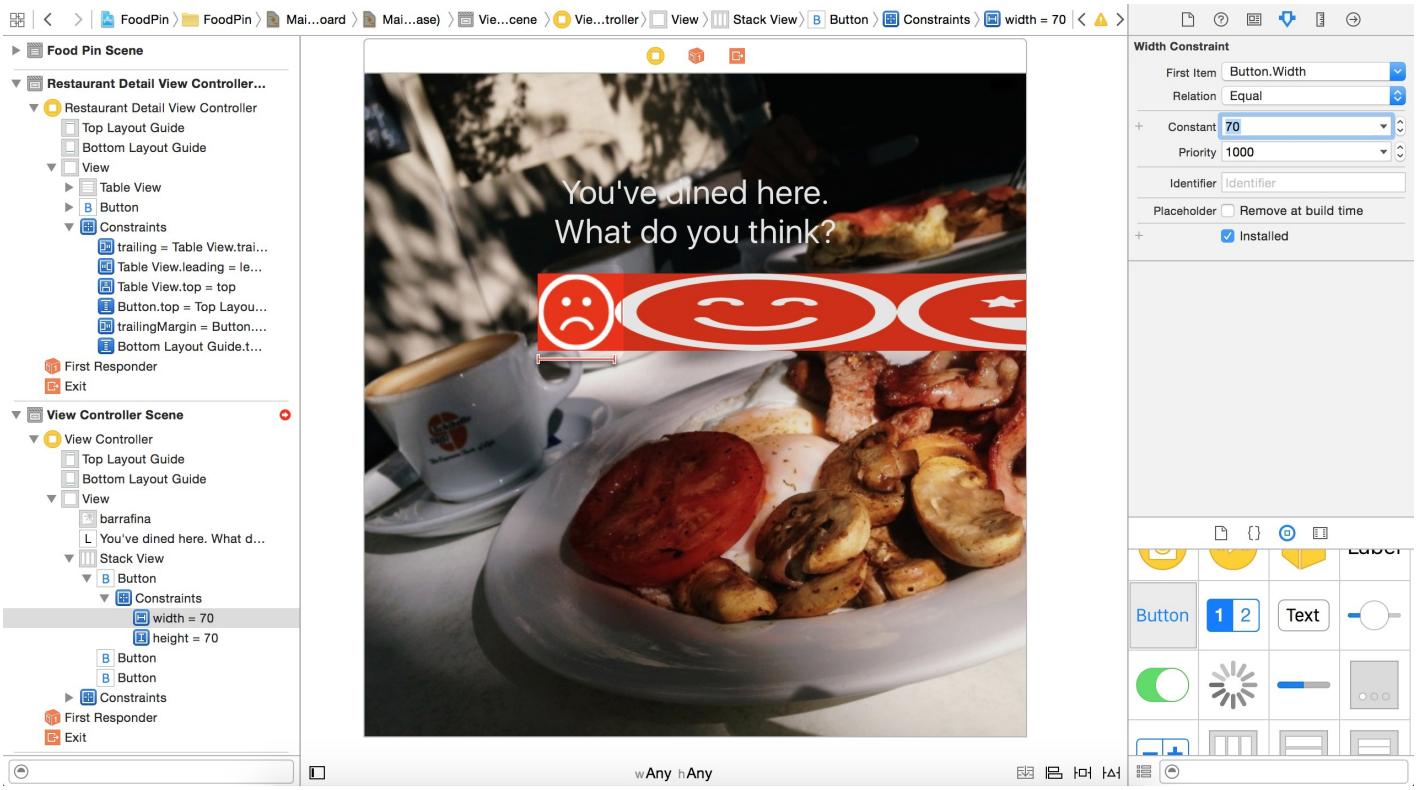

Figure 16-8. Editing the width and height constraints

Repeat the same procedures for the other two buttons. Your button layout should be very similar to that in figure 16-9. To add a space between the buttons, select the stack view and change the spacing option to `10` in the Attributes inspector.

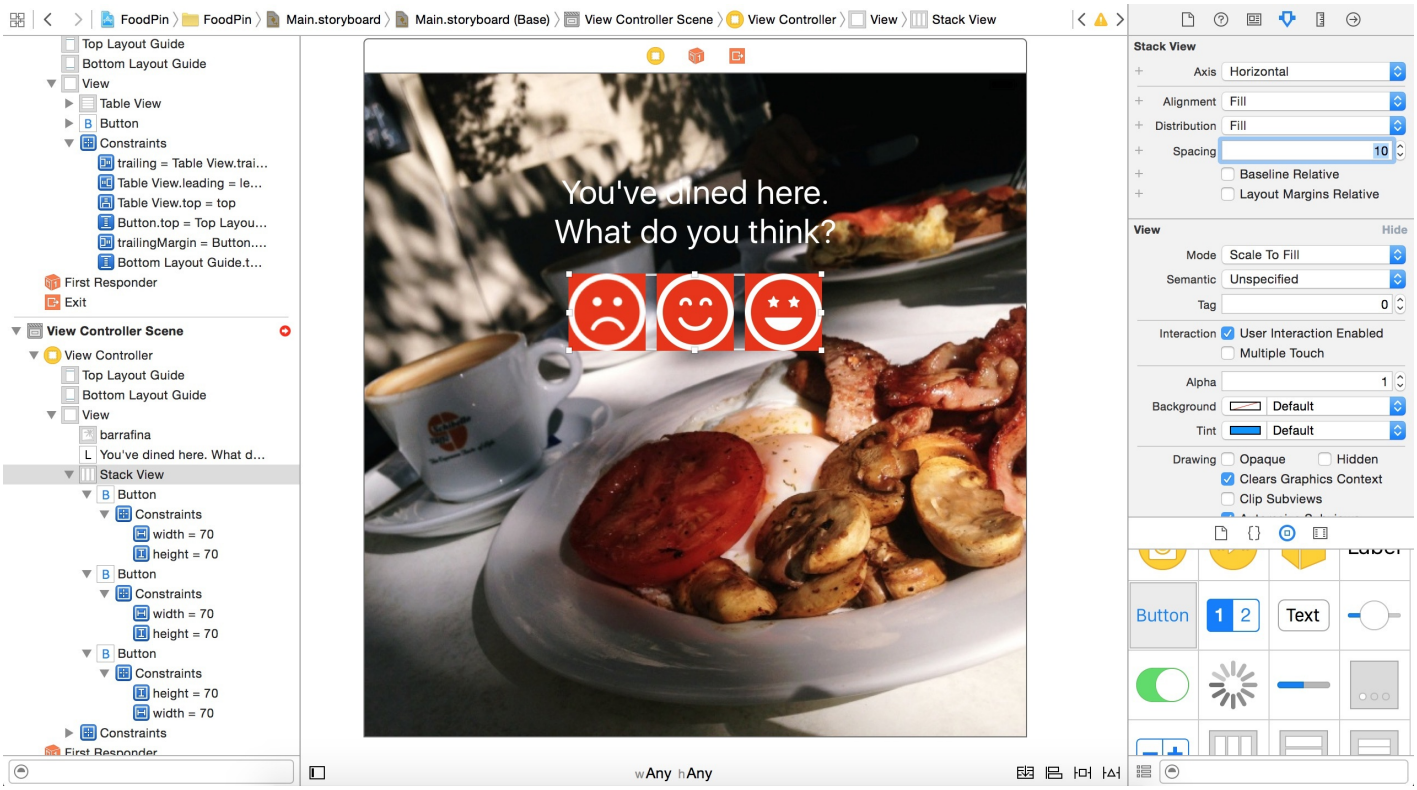

Figure 16-9. The resulting stack view

Again, for this simple layout, you may use the "Add Missing Constraints" option to let Xcode add the constraints for you. Select both the label and the stack view, and then click "Resolve Auto Layout Issues" > "Add Misssing Constraints".

Lastly, add another button and place it near the upper right corner of the view. Set the title to `blank` and image to `close`. In the Size inspector, change its width and height to `30` points. Then click the *Pin* button in the layout bar to add four layout constraints.

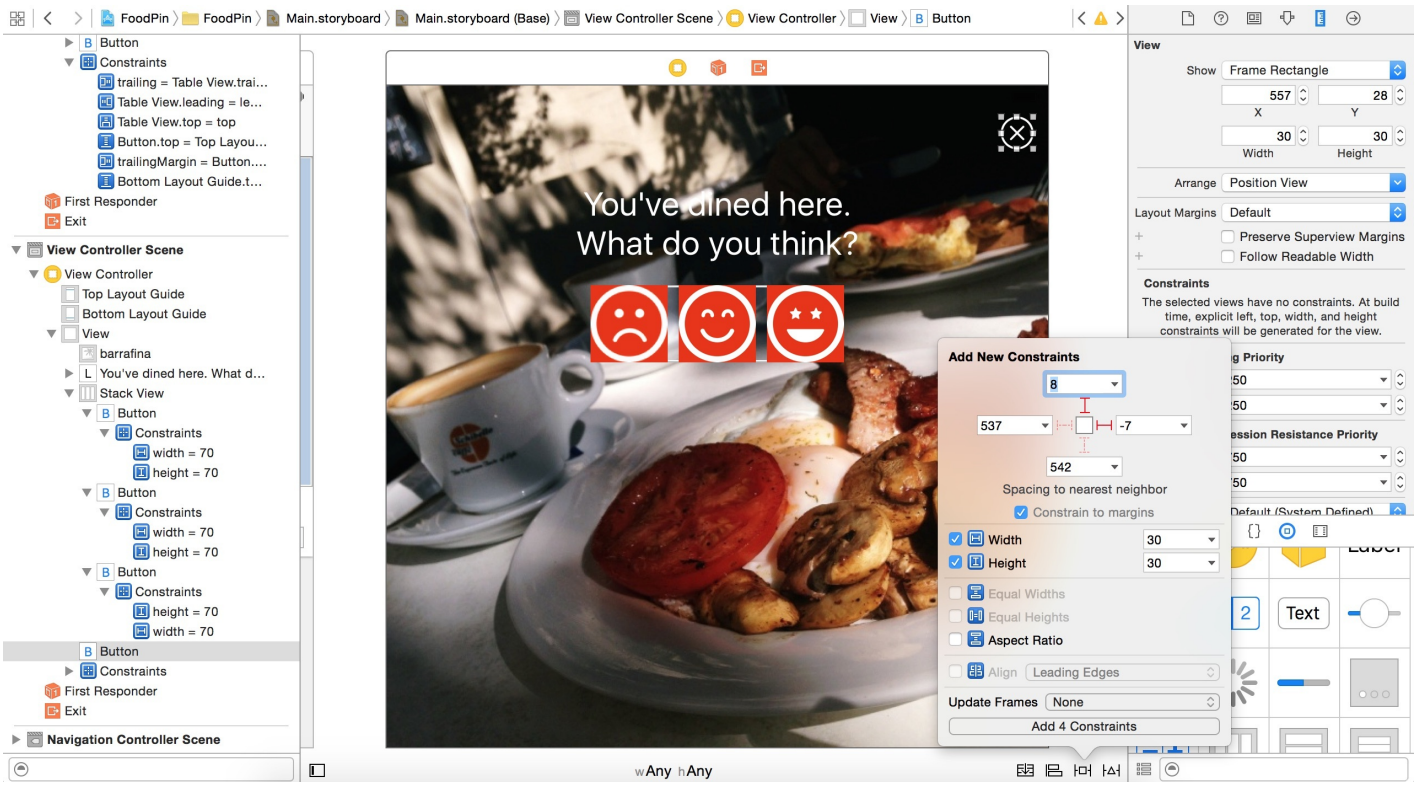

Figure 16-10. Adding layout constraints for the close button

Create a Segue for the Modal View

To bring up the review view modally, we have to connect the *Review* button with the review view controller with a segue. Hold the control key and drag from the *Review* button to the review view controller. Release the buttons and select *Present modally* as the segue type. Once the segue is created, select it and set the identifier of the segue to `showReview` under the Attribute inspector.

Figure 16-11. Create a segue to present the review view controller modally

Now let's have a quick test. Run the app and go to the detail view. Tapping the review button should now bring up the review view.

Defining an Exit for the Review View Controller

Presently, there is no way to dismiss the view to return to the previous screen (i.e. detail view controller). What we are going to do is to define a so-called *unwind segue*. An unwind segue can be used to navigate back through a modal or push segue. In this example, we can use it to dismiss the modal view. To use an unwind segue, you need to do two things. First, declare a method in the destination view controller. In this case, it is the `RestaurantDetailViewController` class. Add the following method in `RestaurantDetailViewController.swift`:

```
@IBAction func close(segue:UIStoryboardSegue) {

}
```

Before you can begin adding unwind segues in Interface Builder, you must define at least one unwind action. This action method tells Xcode that it can be unwound. Optionally, you can implement additional logic in the method. Meanwhile, an empty method is good enough for us to configure an unwind segue. Go back to Interface Builder. Press and hold control key and drag from the *Close* button to the Exit icon of the scene dock (see figure 16-12). When prompted, select `close:` for the action segue.

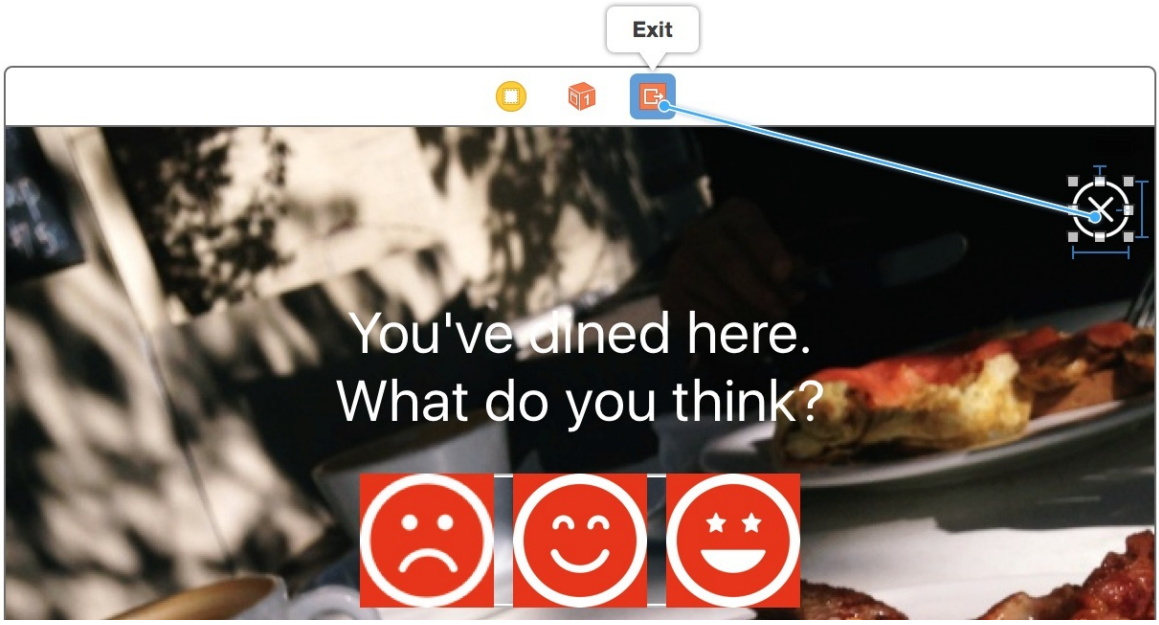

Figure 16-12. Adding an unwind segue for the close button

You can run the project again. Now when the user taps the close button, the modal view will be dismissed.

Applying a Blurring Effect to the Background Image

In iOS 8, Apple introduced a new class called `UIVisualEffectView` that lets developers apply visual effects to a view. Combining with the `UIBlurEffect` class, you can easily apply a blurring effect to an image view.

Now let's see how we can blur the background image. First, create a new class for the review view controller. In the project navigator, right click the `FoodPin` folder and select "New File...". Select the *Cocoa Touch Class* template to proceed. Name the class `ReviewViewController` and set it as a subclass of `UIViewController`. Then save the file in the FoodPin folder.

Select the `ReviewViewController.swift` file you've just created. Because we are going to apply a blurring effect to the image view, let's add an outlet variable for it:

```
@IBOutlet var backgroundImageView:UIImageView!
```

In the `viewDidLoad` method, add the following code:

```
let blurEffect = UIBlurEffect(style: UIBlurEffectStyle.Dark)
let blurEffectView = UIVisualEffectView(effect: blurEffect)
blurEffectView.frame = view.bounds
backgroundImageView.addSubview(blurEffectView)
```

To apply a blurring effect to the background image view, all you need to do is create a `UIVisualEffectView` object with the blurring effect, followed by adding the visual effect view to the background image view. The `UIBlurEffect` class offers three different styles: *dark*, *light*, and *extra light*. I like the dark style but it is up to you to choose your own style. The above lines of code are all you need to blur the background image.

Now, go to Interface Builder and select the review view controller. In the Identify inspector, set the custom class to `ReviewViewController`. Finally, establish a connection between the background image view and the `backgroundImageView` outlet.

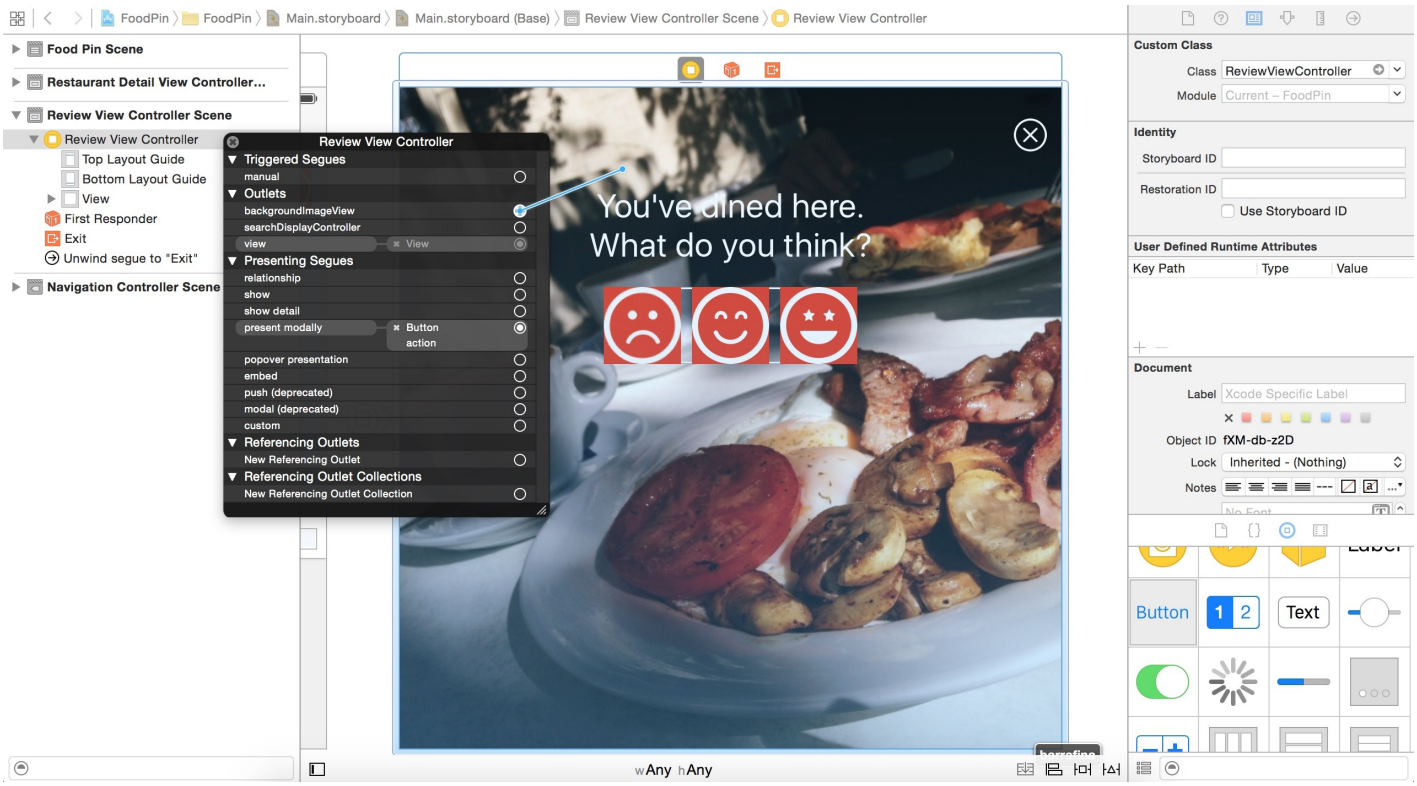

Figure 16-13. Establishing a connection between the outlet variable and the image view

You're ready to go. Figure 16-14 displays the resulting screen after all the changes.

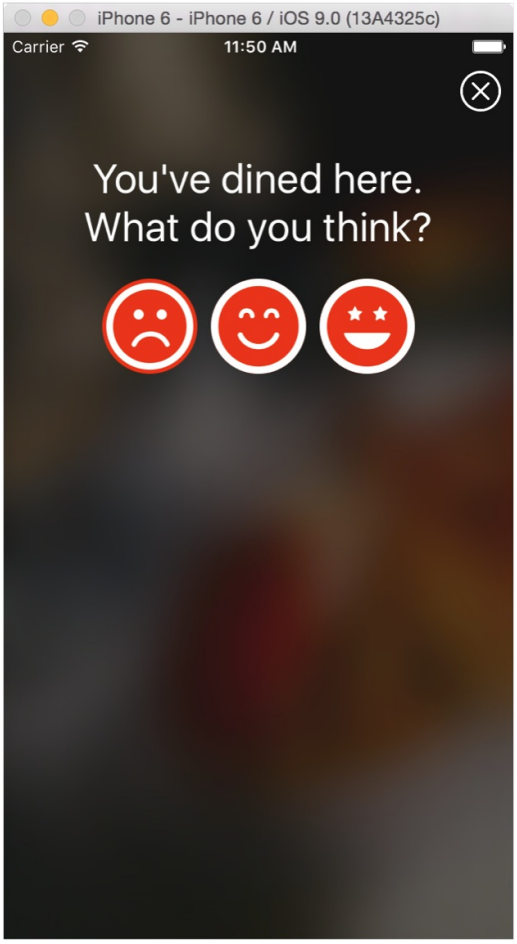

Figure 16-14. Review screen now looks great after applying blurring effect

Animating Dialog View Using UIView Animation

After all the preparation, finally we come to the core part of the chapter: *UIView animation*. We're going to add a growing animation for the stack view. The growing effect is very similar to the growing circle animation we talked about earlier. The stack view does not appear when the modal view is first displayed, but the dialog view starts to grow till it reaches its expected size. Figure 16-15 gives you a better idea of the animation.

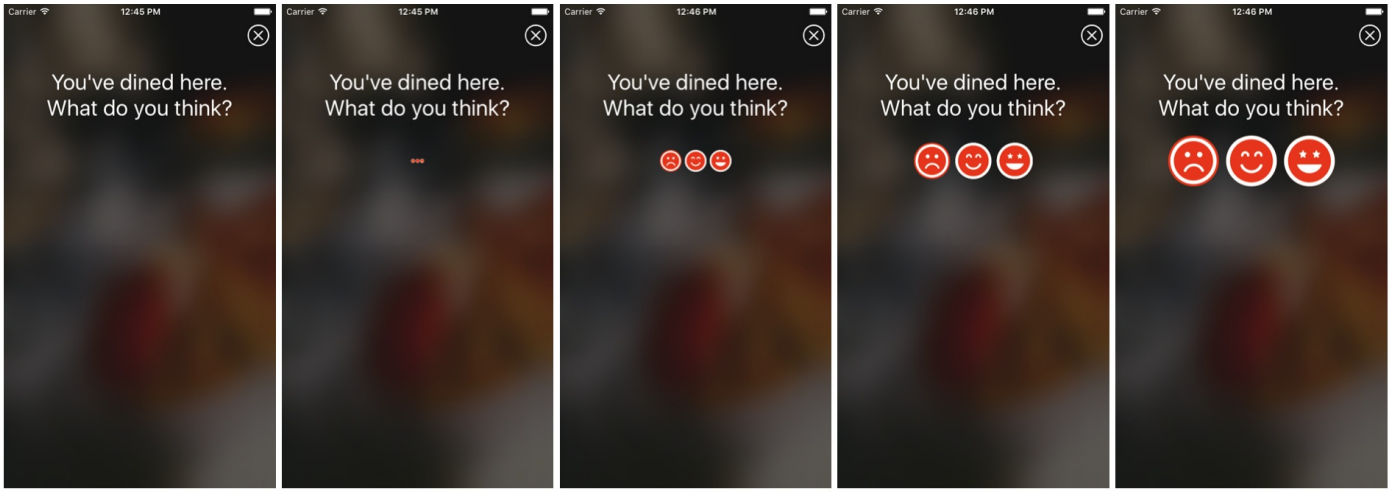

Figure 16-15. Growing animation

As explained before, you just need to provide two states (start and end state) of an animation. `UIView` will then generate the animation for you. For the growing animation, these are the start and end states:

- Start state - the dialog view is in zero size
- End state - the dialog view is in its regular size

At this point you might be trying to figure out how to resize a view. iOS provides transform functions for you to scale, rotate, and move a view. To scale a view, create an affine transformation using `CGAffineTransformMakeScale` and set it to the `transform` property of a `UIView` object.

```
CGAffineTransformMakeScale(0.0, 0.0)
```

In the `ReviewViewController` class, add another outlet variable for the stack view:

```
@IBOutlet var ratingStackView:UIStackView!
```

Go back to Interface Builder and establish the connection between the stack view and the outlet variable.

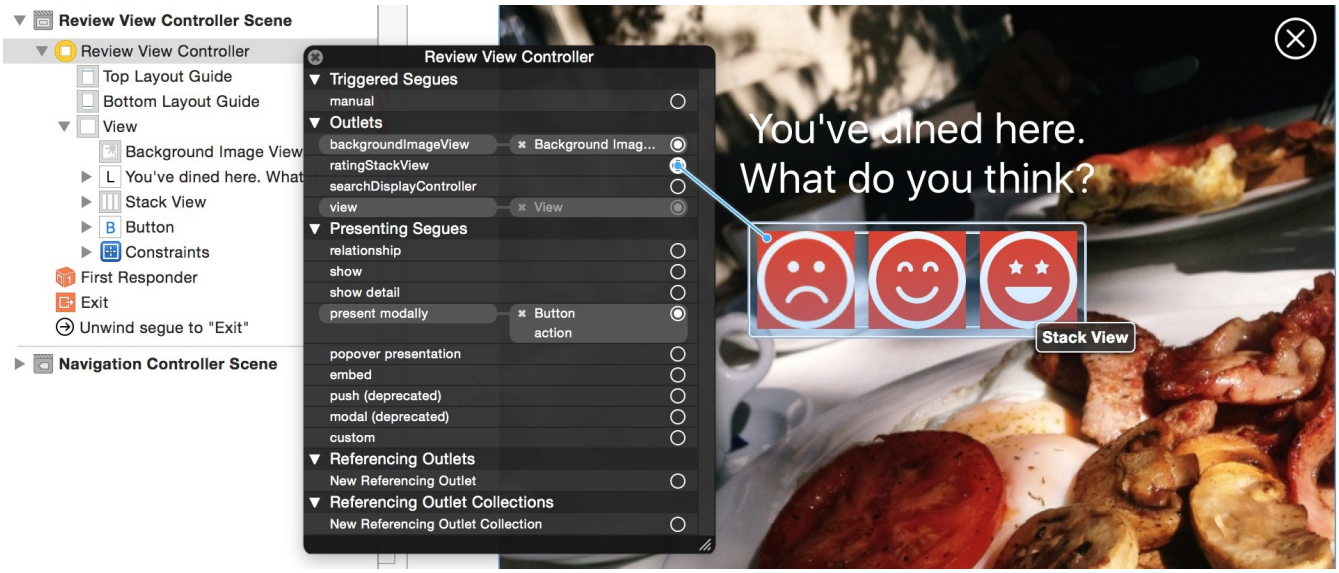

Figure 16-16. Establish a connection between the stack view and the outlet variable

In the `viewDidLoad` method of the `ReviewViewController` class, add the following line of code:

```
ratingStackView.transform = CGAffineTransformMakeScale(0.0, 0.0)
```

This scales down the stack view when it is first loaded. To create the growing effect, we use the `UIView.animationWithDuration` method:

```
override func viewDidAppear(animated: Bool) {

    UIView.animateWithDuration(0.4, delay: 0.0, options: [], animations: {
        self.ratingStackView.transform = CGAffineTransformIdentity
    }, completion: nil)

}
```

As we want to load the animation after the view is loaded, we put the animation block in the `viewDidAppear` method. The duration of the animation is set to 0.7 seconds. In the animations closure, we define the final state of the dialog view. In this case, we simply scale it to the original size. `CGAffineTransformIdentity` is a constant for resetting a view to its original size and position. Figure 16-17 depicts how this code works.

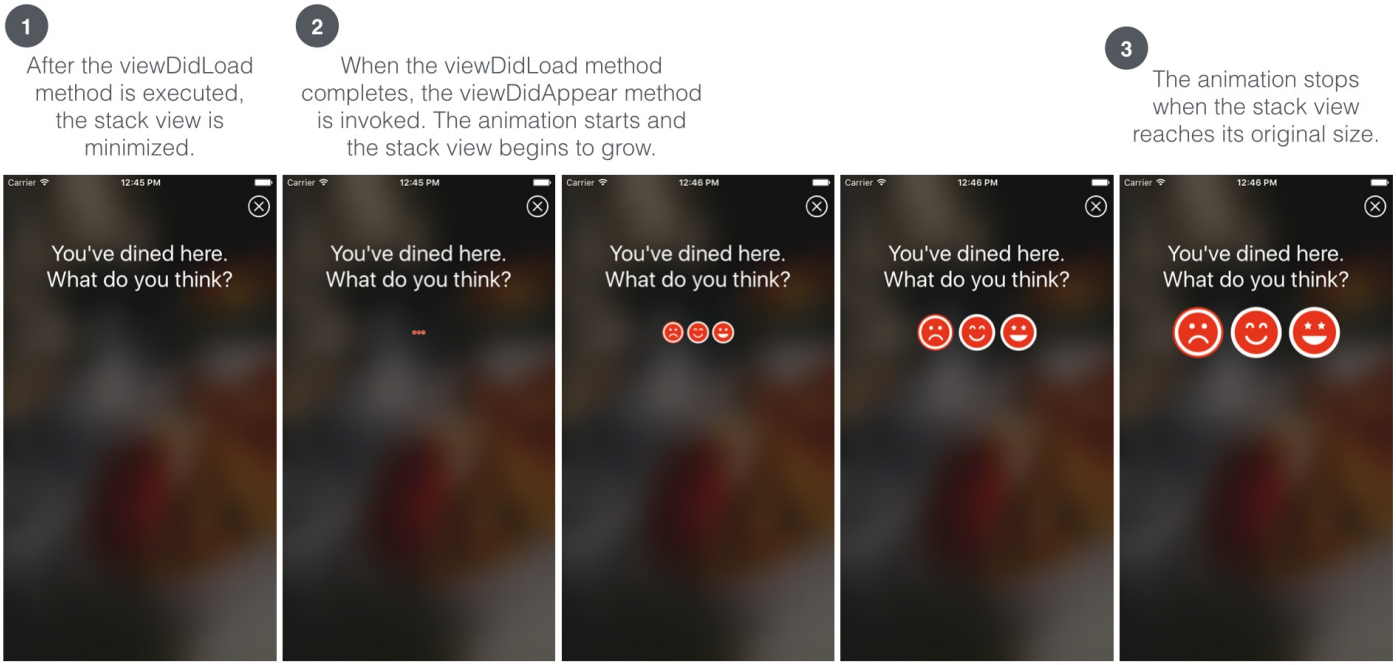

Figure 16-17. How the growing animation works

That's it. UIView automatically renders the animations. Run the app for a quick test and enjoy the animation.

Quick note: You can change the duration parameter from 0.4 to other value (e.g. 2.0) to slow down or speed up the animation.

Spring Animation

The animation is cool, right? Let me introduce a variation of the `UIView` animation known as *spring animation*. Spring animation is very common in iOS 7 or later. One example is the animation during app launch. To take advantage of spring animation in your app, here is the single method call you need:

```
UIView.animateWithDuration(0.5, delay: 0.0, usingSpringWithDamping: 0.3,
initialSpringVelocity: 0.5, options: [], animations: {

    self.ratingStackView.transform = CGAffineTransformIdentity

}, completion: nil)
```

The method should look familiar to you, but it adds the `damping` and `initialSpringVelocity` parameters. Damping takes a value from 0 to 1, and controls how much resistance the spring has when it approaches the end state of an animation. If you want to increase oscillation, set to a lower value. The `initialSpringVelocity` property specifies the initial spring velocity. Try to replace the existing animation code with the above code and see how the spring animation works.

Slide-Up Animation

`CGAffineTransformMakeScale` is one of the common transforms for scaling a view. Let's use another affine transform to create a slide-up animation:

```
CGAffineTransformMakeTranslation(x, y)
```

`CGAffineTransformMakeTranslation` allows you to change the view's position. For a slide-up animation, we first move the stack view off screen and then bring it back to its original position. In the `viewDidLoad` method, add the following line of code:

```
ratingStackView.transform = CGAffineTransformMakeTranslation(0, 500)
```

The line of code will move the stack view off screen (bottom). The final state is to restore the stack view to the original position. So we can keep the `viewDidAppear` method intact.

Now run the app again and the slide-up animation should work.

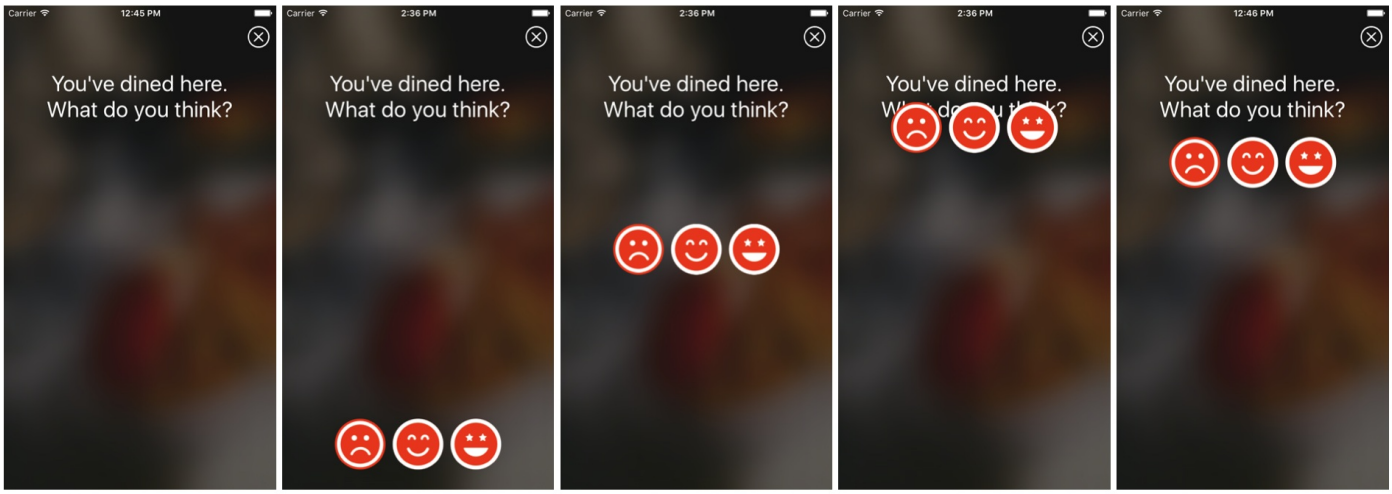

Figure 16-18. Slide-up animation

Combining Two Transforms

What's interesting about affine transform is that you can combine two transforms together. This is the function you need to remember:

```
CGAffineTransformConcat(transform1, transform2)
```

Now that we have both the *scale* and *translate* transform. Wouldn't it be great if we can combine them together? In the `viewDidLoad` method, replace the following code:

```
ratingStackView.transform = CGAffineTransformMakeTranslation(0, 500)
```

with:

```
let scale = CGAffineTransformMakeScale(0.0, 0.0)
let translate = CGAffineTransformMakeTranslation(0, 500)
ratingStackView.transform = CGAffineTransformConcat(scale, translate)
```

The code is very straightforward. We combine the two transforms using `CGAffineTransformConcat`. Again, we can keep the `viewDidAppear` method unchanged.

That's it! You're ready to go and see how the combined animation works.

Unwind Segues and Data Passing

Earlier, we used an unwind segue to "go back" to the detail view, when a user taps the *close* button. How about the *rating* buttons? How can we pass the selected rating from the review view controller to the detail view controller so that it can update its rating image?

We will make use of the unwind segue to faciliate the data passing. Here are what we are going to implement:

- We will add an action method in the `ReviewViewController` class - when any of the buttons (dislike/good/great) is tapped, the method will be called.

- In the method, we determine which button is tapped, and save the corresponding rating in a property. It then initiates the unwind segue to "go back" to the detail view.
- The `close` action method will be called. Here we can get the source view controller through the `UIStoryboardSegue` object, and retrieve the rating from the controller.
- Then we can update the rating image of the detail view.

It sounds complicated, but the implementation is quite similar to the one we have worked on in chapter 12.

Let's first give the unwind segue an identifier because we will use it to trigger the segue in the code. In Main.storyboard, choose the unwind segue and set the identifier to *unwindToDetailView* in the Identity inspector.

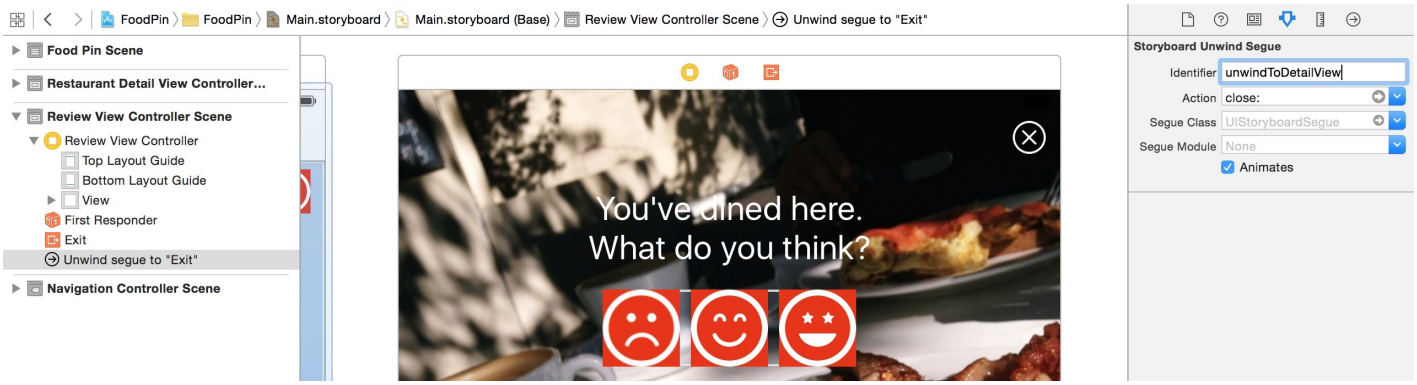

Figure 16-19. Adding an identifier for the unwind segue

Because we need to update the rating button of the detail view, we have to associate it with an outlet. In the `RestaurantDetailViewController` class, declare an outlet variable:

```
@IBOutlet var ratingButton:UIButton!
```

In Interface Builder, right click the Restaurant Detail View Controller and connect the `ratingButton` outlet to the rating button.

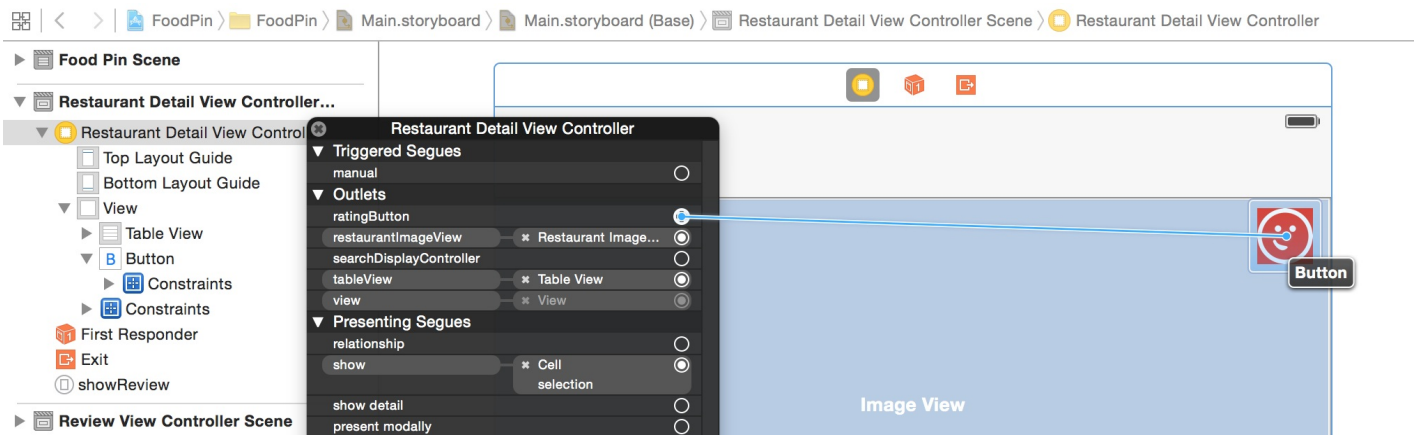

Figure 16-20. Connect the rating button with the outlet

In order to differentiate one button to another, you can assign a tag to each of the buttons. In the Interface Builder editor, select the *dislike* button and go to the Attributes inspector. Set the tag value to `100`. The *tag* option lets you define an integer to identify the button (or other view objects). Repeat the same procedures for the *good* and *great* buttons. Set the tag value to `200` and `300` respectively.

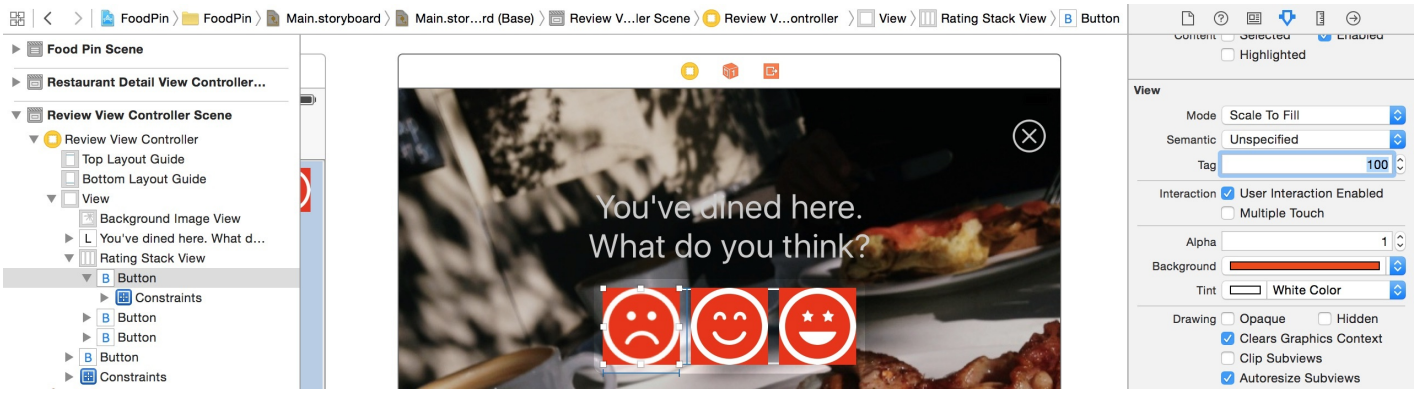

Figure 16-21. Assign a tag for the button

With the UI objects configured, it's now ready to create the action method. This action method is called when one of the rating button is tapped. Let's name the method `ratingSelected` and take in a `UIButton` object. Forget about the implementation, we will talk about it in a minute.

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

Now go back to `Main.storyboard`. Control-drag from the *dislike* button to the Review View Controller. Release the buttons and choose `ratingSelected:` to connect with the action method. Repeat the same procedures for the *good* and *great* buttons.

We're going to implement the action method. But first let's declare a `rating` property in the `ReviewViewController` class to store the selected rating (dislike/good/great):

```
var rating:String?
```

As mentioned before, we have to determine which button is tapped in the action method, and trigger the unwind segue. The real question is how. When the action method is triggered, the selected button will be passed through the `sender` parameter. We can access the `tag` property of the UIButton object and map it to the corresponding rating. To trigger a segue, all you need to do is invoke the `performSegueWithIdentifier` method with the corresponding segue identifier. So you can implement the action method like this:

```
@IBAction func ratingSelected(sender: UIButton) {
    switch (sender.tag) {
    case 100: rating = "dislike"
    case 200: rating = "good"
    case 300: rating = "great"
    default: break
    }

    performSegueWithIdentifier("unwindToDetailView", sender: sender)
}
```

Once the unwind segue is triggered, the `close` method in the `RestaurantDetailViewController` method will be called. You can update the method like below to update the image of the rating button accordingly. Here we get the source view controller from the `segue` object, find out the selected rating and then set the corresponding image of the rating button.

```
@IBAction func close(segue:UIStoryboardSegue) {
    if let reviewViewController = segue.sourceViewController as?
ReviewViewController {
        if let rating = reviewViewController.rating {
            ratingButton.setImage(UIImage(named: rating), forState:
UIControlState.Normal)
        }
    }
}
```

That's it! You've passed the selected rating from one controller to another. Hit the Run button to test the app. Choose a restaurant, rate it and the detail view's rating button should be updated accordingly.

Your Exercise

This time, I have two exercises for you. The first one is designed to test your knowledge of Object Oriented Programming. As of now, the rating of a restaurant is not saved in the data model. Try to update the project to store the rating in the `Restaurant` array.

```
Hint: Add a rating property in the Restaurant class.
```

The next exercise is related to animation. Your exercise is to modify the animation of the rating buttons in the Review View Controller. Instead of animating the stack view, try to animate each button and create an animation similar to that shown in figure 16-22.

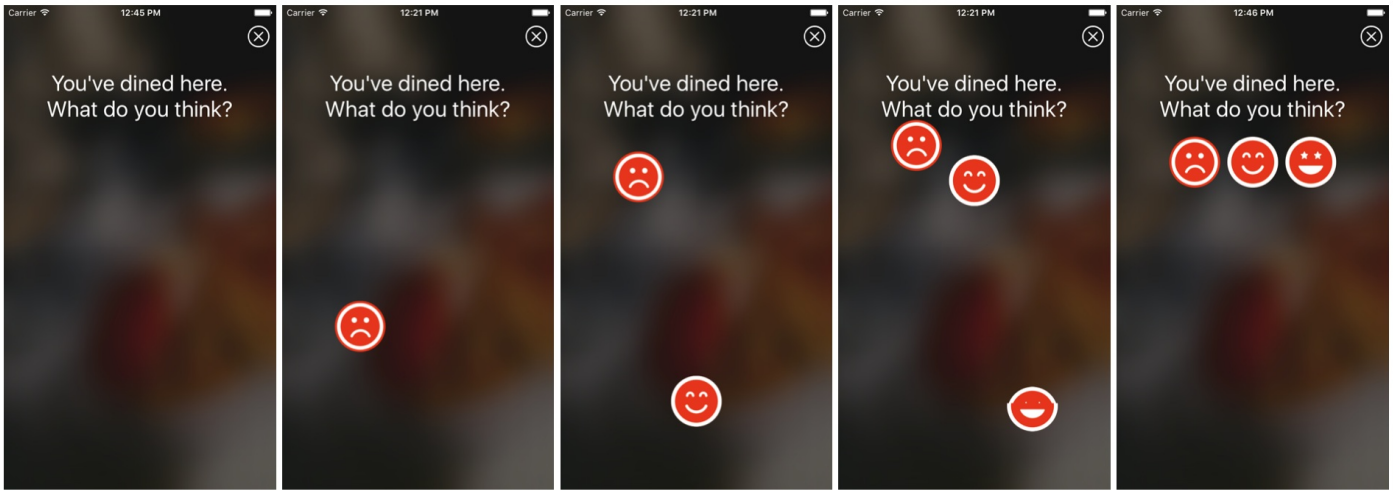

Figure 16-22. Animating the rating buttons

```
Hint: You will need to create an outlet for each of the rating buttons.
```

Summary

This is another huge chapter that covers UIView animation, visual effects, and unwind segues.

I hope you love all the techniques you've learned about UIView animations and visual effects. As you can see, it is super easy to animate a view, whether it's a button or label. I would encourage you to play around with the parameters (like damping, initial spring velocity and delay) and see what animations you can create. And, don't forget to take some time to complete the exercises.