CHAPTER 25

SELF SIZING CELL AND DYNAMIC TYPE

INTRODUCTION TO SELF SIZING CELL AND DYNAMIC TYPE

In iOS 8, Apple introduces a new feature for UITableView known as Self Sizing Cells. To me, this is seriously one of the most exciting features for the new SDK. Prior to iOS 8, if you wanted to display dynamic content in table view with variable height you would need to calculate the row height manually. Now with iOS 8, self sizing cells provide a solution for displaying dynamic content. In brief, here is what you need to do when using self sizing cells:

- Define auto layout constraints for your prototype cell
- Specify the estimatedRowHeight of your table view
- Set the rowHeight of your table view to UITableViewAutomaticDimension

If we express the last point in code, it looks like this:

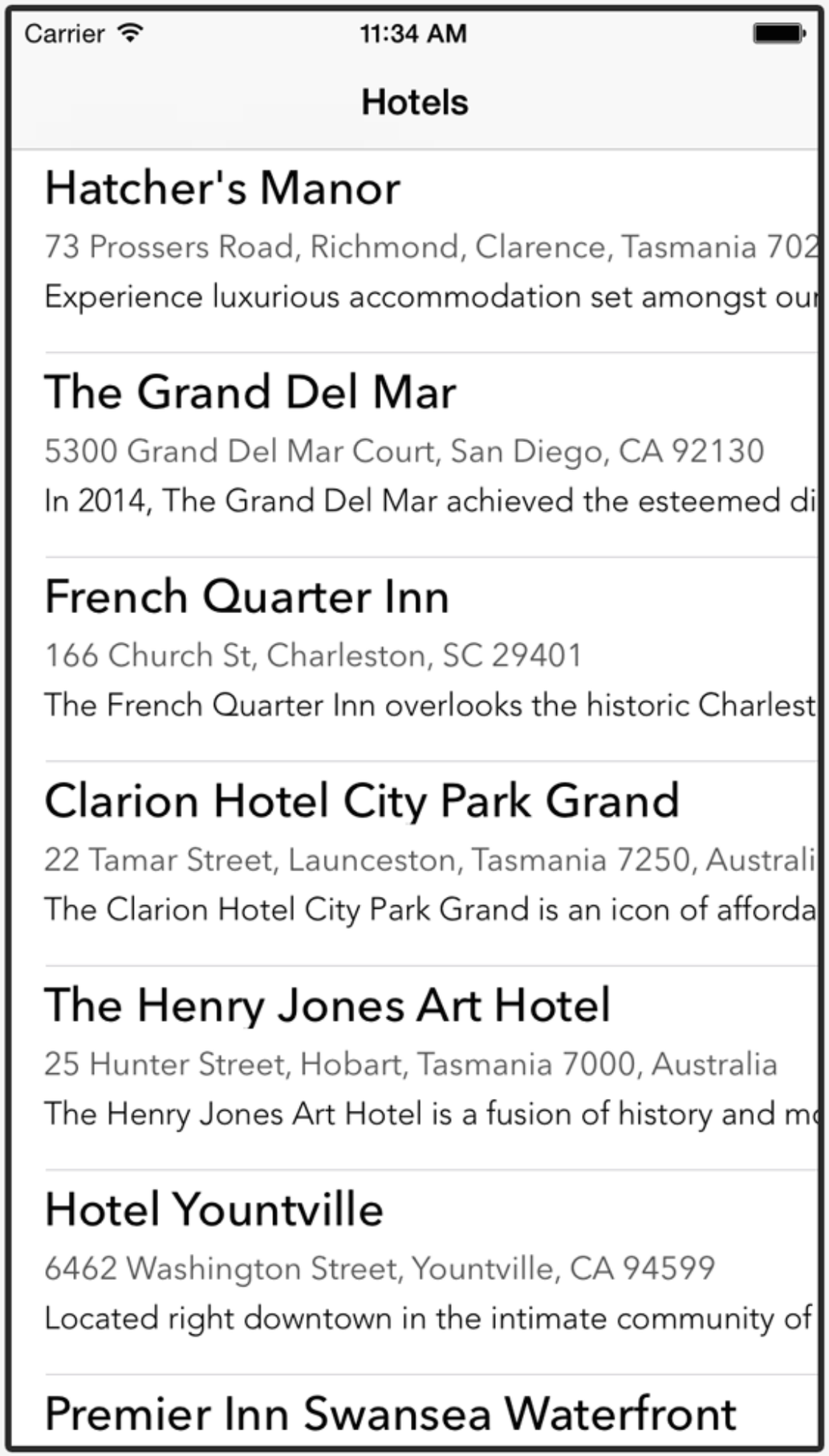
```
tableView.estimatedRowHeight = 95.0
tableView.rowHeight = UITableViewAutomaticDimension
```

With just two lines of code, you instruct the table view to calculate the cell's size to match its content and render it dynamically. This self sizing cell feature should save you tons of code and time. You're going to love it.

In the next section we'll develop a simple demo app to demonstrate self sizing cell. There is no better way to learn a new feature than to use it.

In addition to self sizing cell, I will also talk about Dynamic Type. Dynamic Type was first introduced in iOS 7 - it allows users to customise the text size to fit their own needs. However, only apps that adopt Dynamic Type respond to the text change. As of right now, only a fraction of third-party apps have adopted the feature.

Starting with iOS 8, Apple wants to encourage developers to adopt Dynamic Type. In the later section you will learn how to adopt dynamic type in your apps to improve its user experience for vision-challenged users.



BUILDING A SIMPLE DEMO

We will start with a project template for a simple table-based app showing a list of hotels. The prototype cell contains three one-line text labels for the name, address and description of a hotel.

If you download the project from https://www.dropbox.com/s/7itd2bt7kd31b6t/SelfSizingCellStart.zip?dl=0 and run it, you will have an app like the one shown on the left.

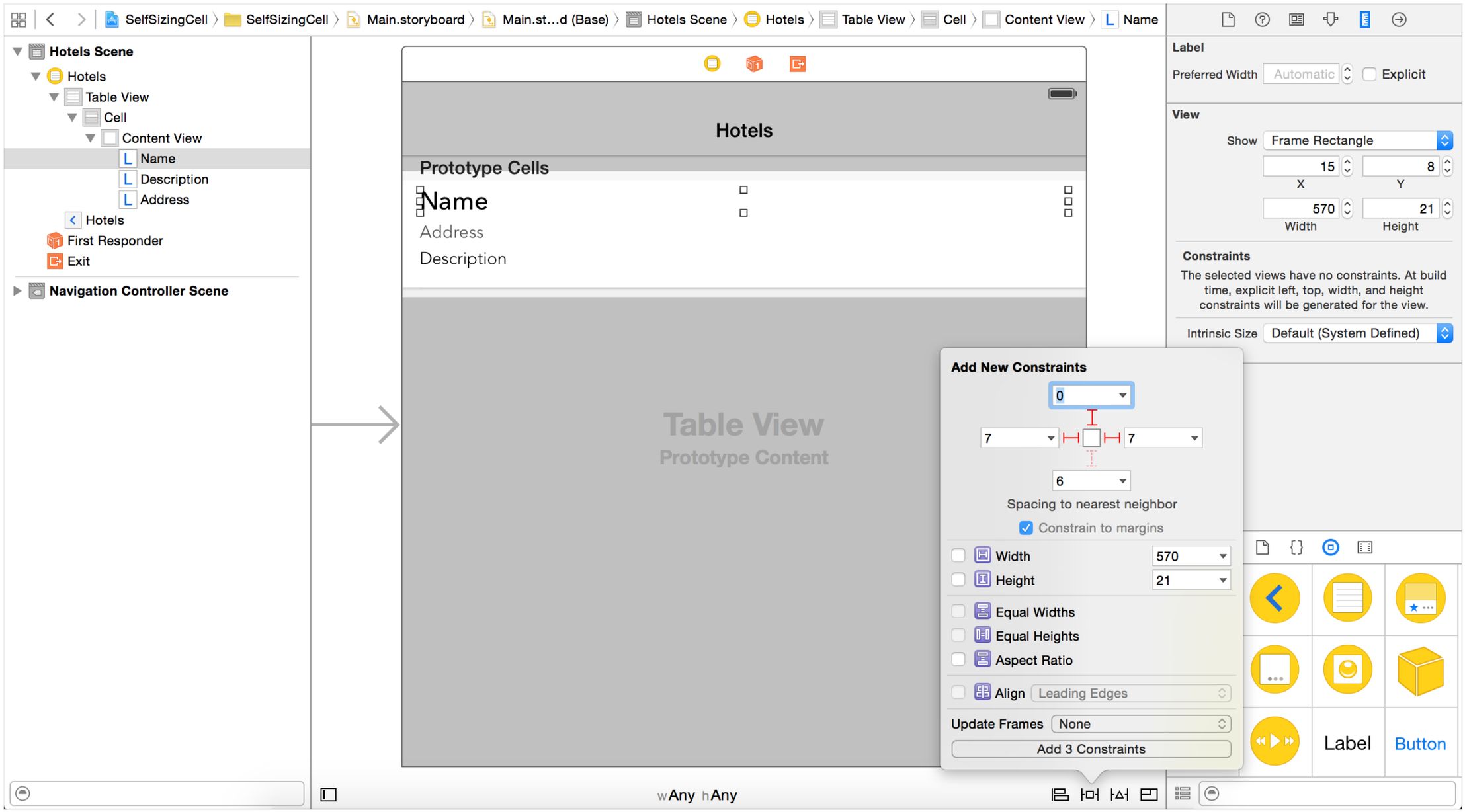
As you can see, some of the addresses and descriptions are truncated; you may have faced the same issue when developing table-based apps. To fix the issue, one quick option is to simply reduce the font size or increase the number of lines of a label. However, this solution is not perfect. As the length of the addresses and descriptions varies, it will probably result in an imperfect UI with redundant white spaces. A better solution is to adapt the cell size with respect to the size of its inner content. Prior to iOS 8, you would need to manually compute the size of each label and adjust the cell size accordingly, which would involve a lot of code and subsequently a lot of time.

Starting from iOS 8, all you need to do is use self sizing cells and the cell size can be adapted automatically. Currently, the project template creates a prototype cell with a fixed height of 95 points. What we are going to do is turn the cells into self sizing cells so that the cell content can be displayed perfectly.

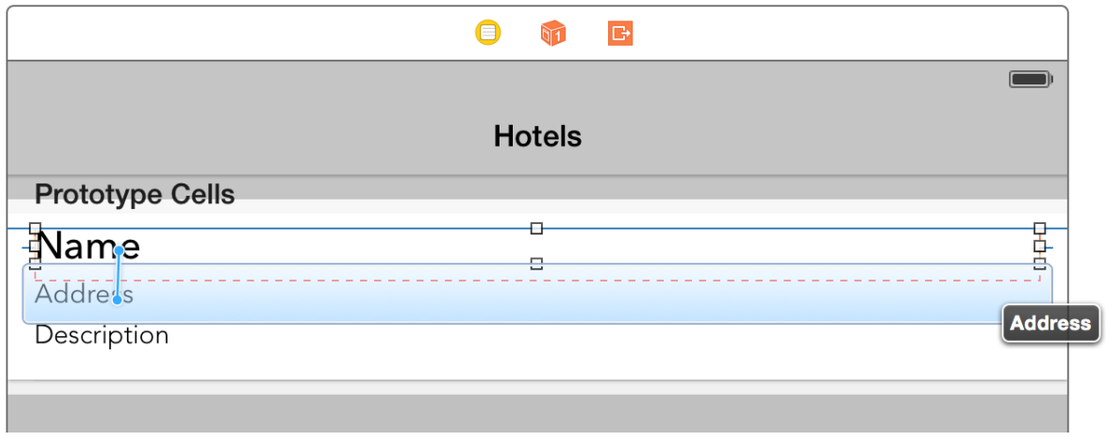
ADDING AUTO LAYOUT CONSTRAINTS

Many developers hate auto layout and avoid using it whenever possible. However, without auto layout self sizing cells will not work, as they rely on the constraints to determine the proper row height. In fact, table view calls *systemLayoutSizeFittingSize* on your cell and that returns the size of the cell based on the layout constraints. If this is the first time you're working with auto layout, I recommend that you quickly review chapter 1 about adaptive UI before continuing.

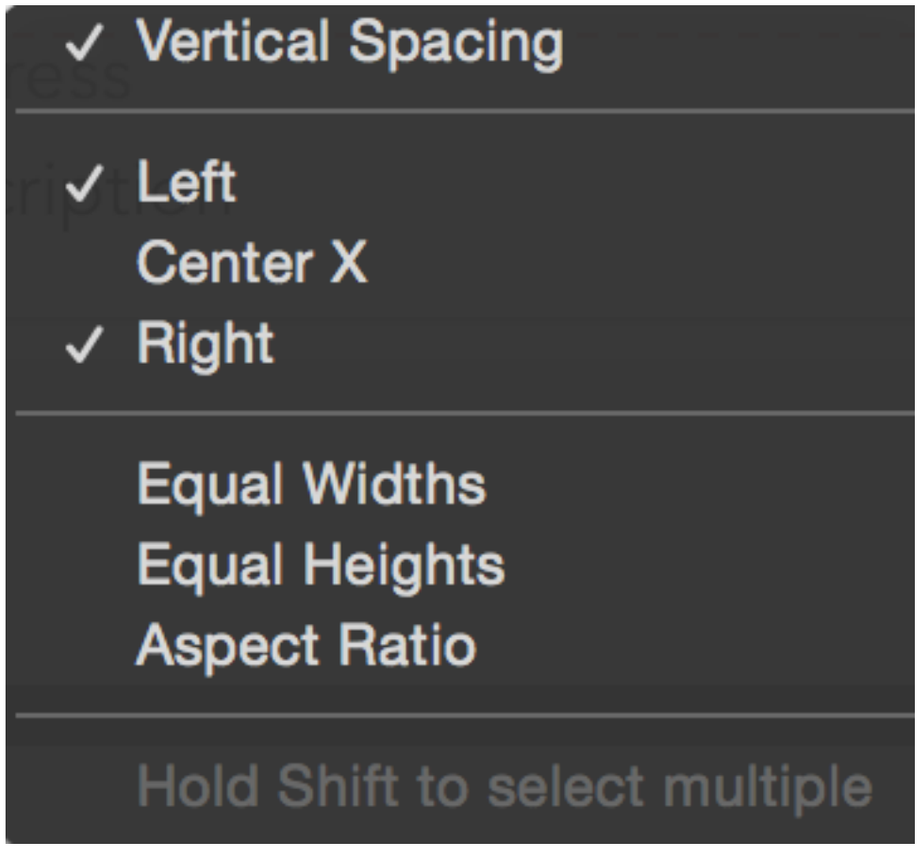
For the project template I did not define any auto layout constraints for the prototype cell; let's add a few constraints to the cell before beginning. Select the name label and click the Pin button of the auto layout menu. Add three spacing constraints for the top, left, and right sides.



Next, control-drag from the name label to the address label.



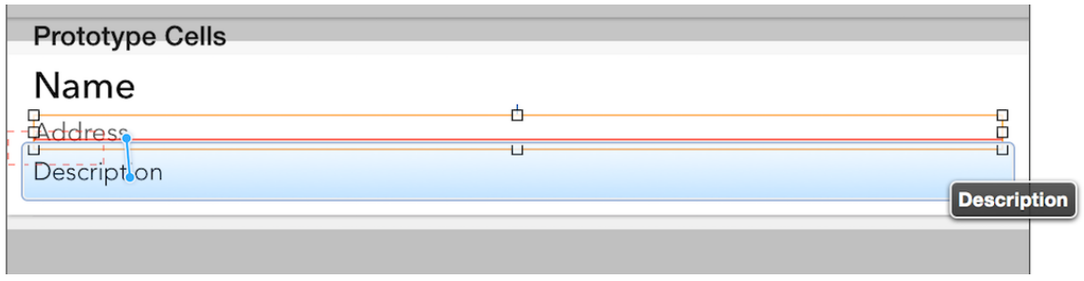
Once you release the button, Xcode shows a context menu. Hold down the shift key and select the “Vertical Spacing”, “Left” and “Right” options.



This defines three layout constraints between the name and address labels:

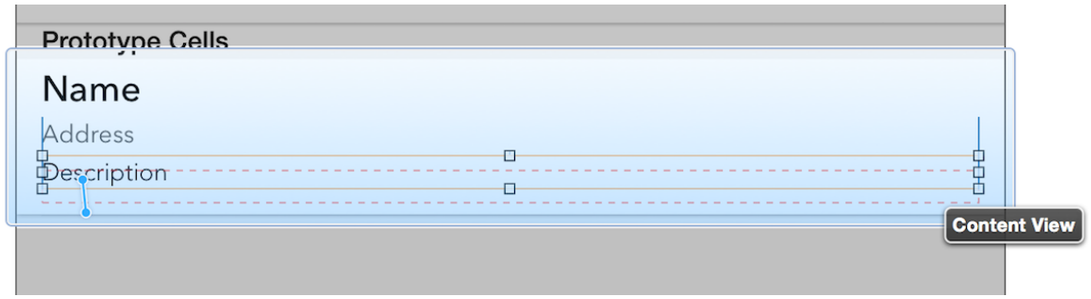
1. A vertical spacing constraint between the name and address labels. For example, the name label should be five points away from the address label.
2. The left side of the address label should align with that of the name label.
3. The right side of the address label should align with that of the name label.

Similarly, control-drag from the address label to the description label.

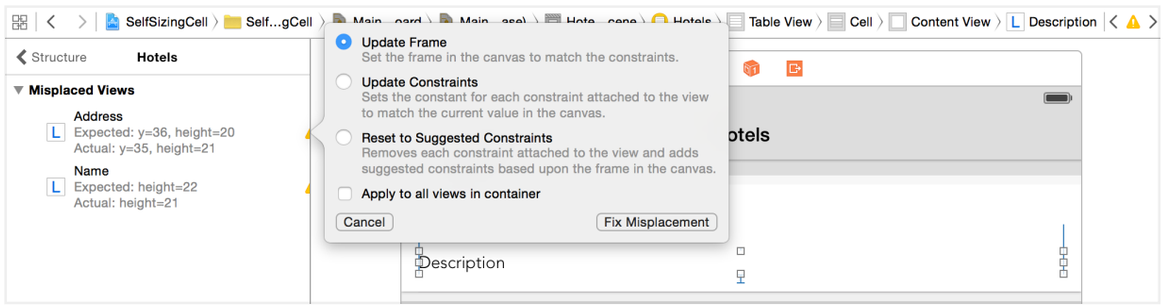


Again, select the “Vertical Spacing,” “Left,” and “Right” options in the context menu. This defines a similar set of layout constraints for the address and description labels.

Lastly, we have to define a spacing constraint between the description label and the bottom part of the cell’s content view. Control-drag from the description label to the content view of the cell.



Once releasing the button, select the “Bottom space to container margin” option. After you configured all the layout constraints, Xcode should detect several auto layout issues. Click the disclosure arrow in the Interface Builder outline view and you will see a list of the issues. Click the warning or error symbol to fix the issue (either by adding the missing constraints or updating the frame).



If you have configured the constraints correctly, your final layout should look similar to this:

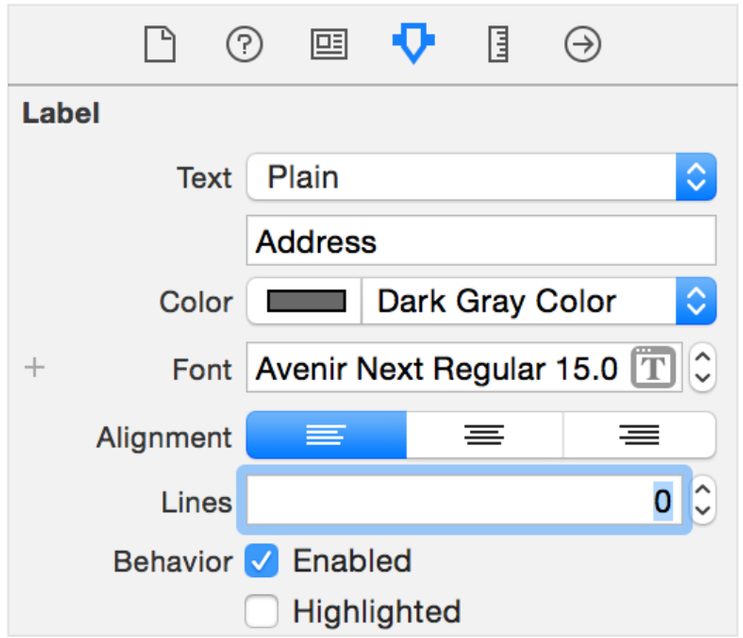
SETTING ROW HEIGHT

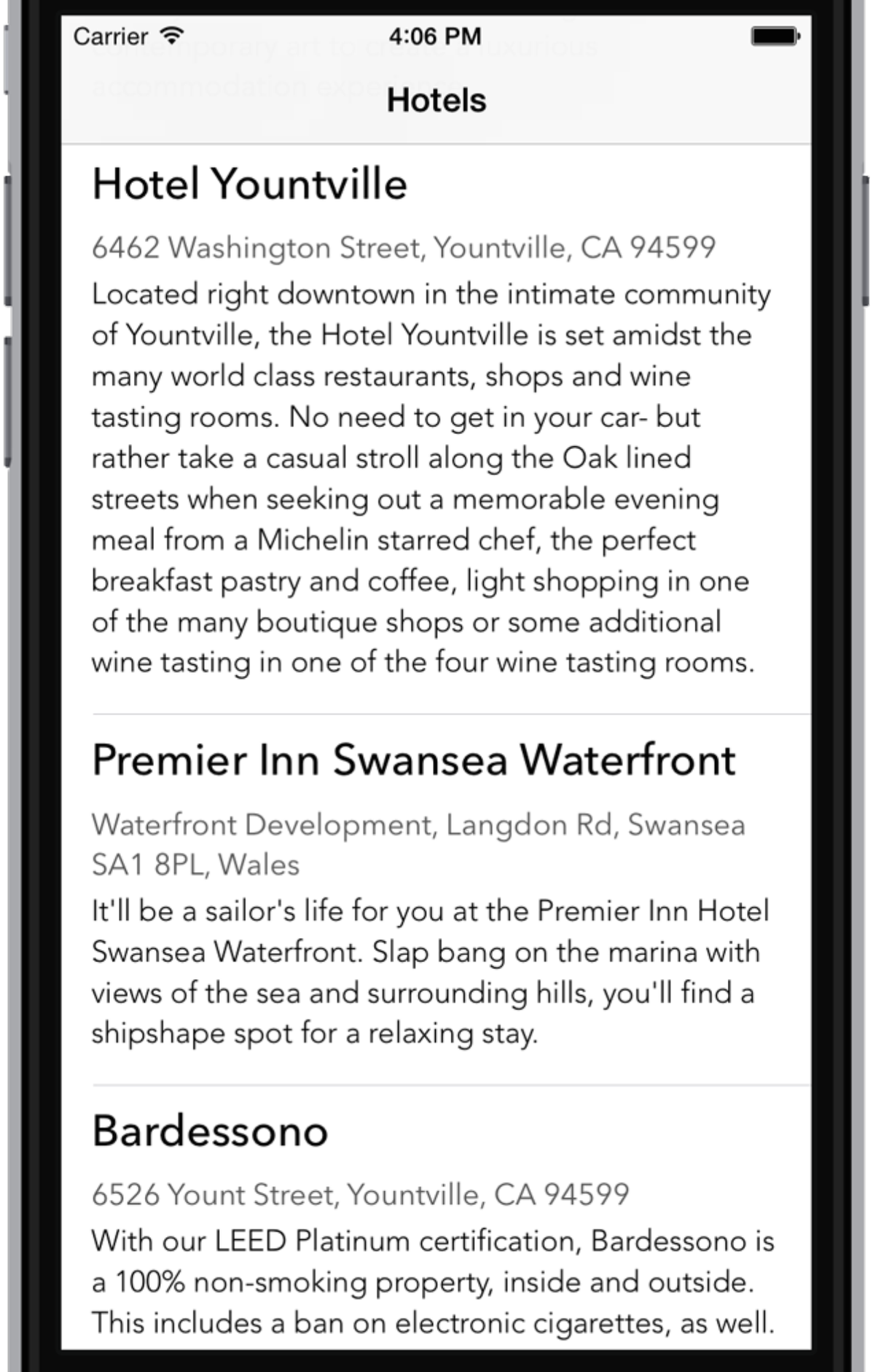
With the auto layout configured, you now need to add the following code in the *viewDidLoad* method of HotelTableViewController:

```
tableView.estimatedRowHeight = 95.0
tableView.rowHeight = UITableViewAutomaticDimension
```

The lines of code set the estimatedRowHeight to 95 points, which is the current row height, and the *rowHeight* property to *UITableViewAutomaticDimension*, which is the default row height in iOS 8. In other words, you ask table view to figure out the appropriate cell size based on the provided information.

If you test the app now, the cells are still not resized. This is because all labels are restricted to display one line only. Select the name label and set the number of lines under the attributes inspector to zero. By doing this, the label should now adjust itself automatically. Repeat the same procedures for both the address and description labels.





Once you made the changes, you can run the project again. This time the cells should be resized properly with respect to the content.

However, there's still a bug when the app is launched; when the table view is first displayed, you may find some of the cells are not sized properly. But when you scroll through the table view, the new cells are displayed with correct row height. To workaround this issue, you can force a table reload after the view appears. First, declare a Boolean variable in the HotelTableViewController class:

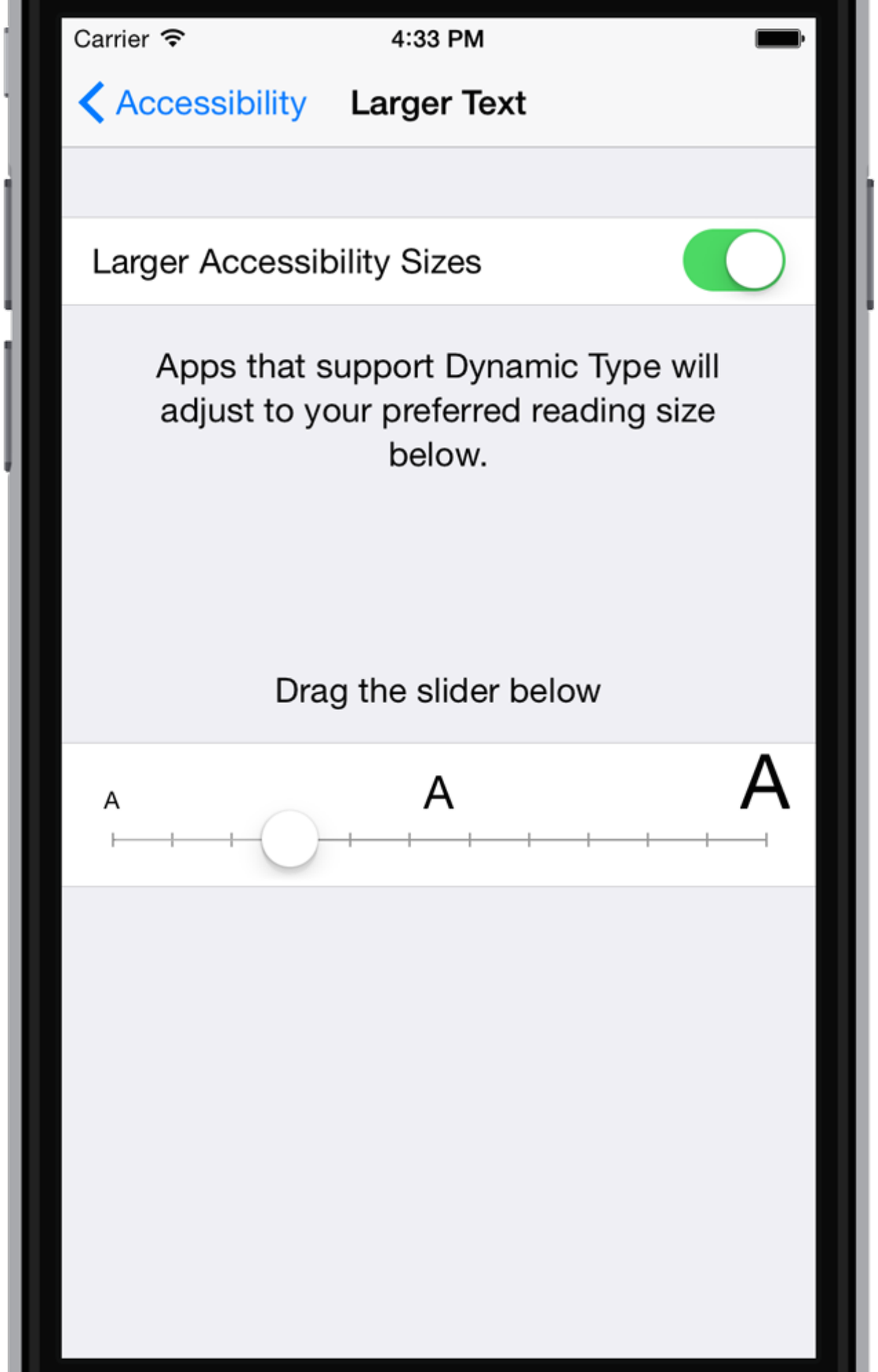
```
var isFirstLoaded = true
```

Then insert the following method:

```
override func viewDidAppear(animated: Bool) {
    if isFirstLoaded {
        let indexSet = NSIndexSet(index: 0)
        tableView.reloadSections(indexSet, withRowAnimation: .None)
        isFirstLoaded = false
    }
}
```

Here we just reload the first section of the table. Because we just want to reload the table once, we set the *isFirstLoaded* variable to *false* as soon as the table is reloaded. Test the app again. It should work now.

For your reference, you can download the final project from https://www.dropbox.com/s/zqpdw0kgkolju9g/SelfSizingCell.zip?dl=0.



DYNAMIC TYPE INTRODUCTION

Self sizing cells are particularly useful to support Dynamic Type. You may not have heard of Dynamic Type but you probably see the setting screen (Settings > General > Accessibility > Larger Text) shown on the left.

Dynamic Type was first introduced in iOS 7 - it allows users to customise the text size to fit their own needs. However, only apps that adopt dynamic type respond to the text change. I believe most of the users are not aware of this feature because only a fraction of third-party apps have adopted the feature.

Beginning with iOS 8, Apple wants to encourage developers to adopt Dynamic Type. As mentioned in the WWDC, all of the system applications have adopted Dynamic Type and the built-in labels automatically have dynamic fonts. When the user changes the text size, those labels are also going to change size.

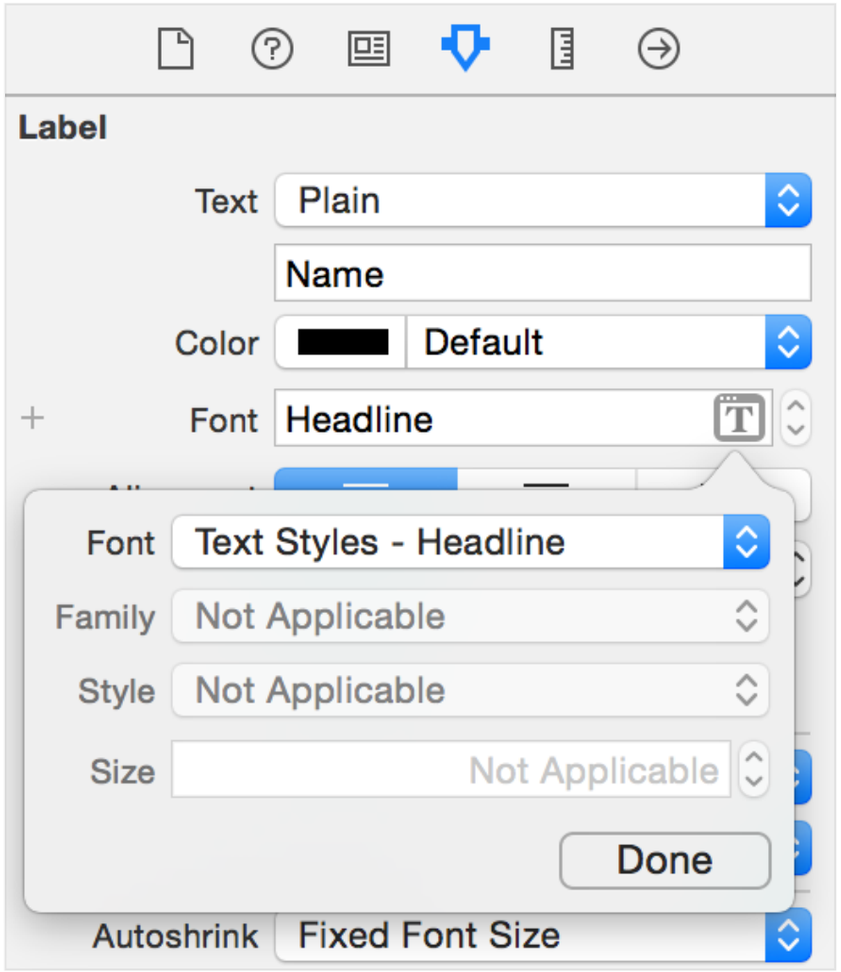
Furthermore, the introduction of Self Sizing Cells is a way to facilitate the widespread adoption of Dynamic Type. It saves you a lot of code from developing your own solution to adjust the row height. Once the cell is self-sized, adopting Dynamic Type is just a piece of cake.

Let's explore how to apply dynamic type to the demo app.

ADOPTING DYNAMIC TYPE

Change the font of the label in the demo project from a custom font to a preferred font for text style (i.e. headline, body, etc).

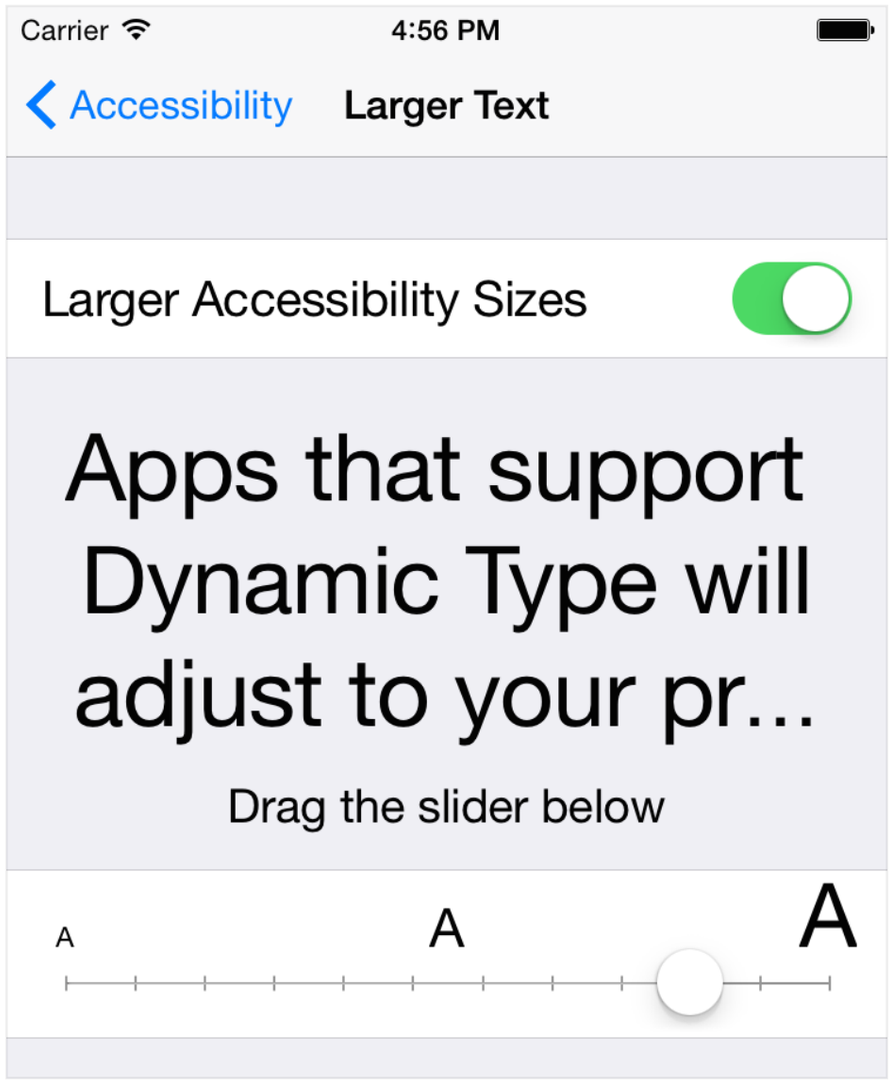
First, select the name label and go the attributes inspector. Change the font option to *Headline*.



Next, select the address label and change the font to *Subhead*. Repeat the same procedure but change the font of description label to *Body*.

As the font style is changed, Xcode should detect some auto layout issues. Just click the disclosure indicator on the Interface Builder outline menu to fix the issues.

That's it. Before testing the app, you should first change the text size. In the simulator, go to Settings > General > Accessibility > Larger Text and enable the Larger Accessibility Sizes option. Drag the slider to set to your preferred font size.



Now run the app and it should adapt to the text size change.

RESPONDING TO TEXT SIZE CHANGE

Now that the demo app has adopted dynamic type, but it doesn't respond to text size change yet. While running the demo app, go to Settings and change the text size. When you switch back to the app, the font size will not be adjusted accordingly; you must quit the app and re-launch it to effectuate the change.

In order to respond to the size change, the app has to listen to the *UIContentSizeCategoryDidChangeNotification* event and reload the table view accordingly. The notification is posted when the user changes the preferred content size setting.

All you need to do now is add an observer in the *viewDidLoad* method of the HotelTableViewController class:

```
NSNotificationCenter.defaultCenter().addObserver(self,
    selector: "onTextSizeChange:",
    name: UIContentSizeCategoryDidChangeNotification,
    object: nil)
```

The code asks the Notification Center to call the *onTextSizeChange* method when the UIContentSizeCategoryDidChangeNotification is posted. Implement the *onTextSizeChange* method like this:

```
func onTextSizeChange(notification: NSNotification) {
    tableView.reloadData()
}
```

When the text size changes, we simply reload the table view to refresh the content. As we use custom table view cells, we have

to explicitly set the fonts each time when the cells are loaded. Update the *tableView(_:cellForRowAtIndexPath:)* method like this (changes are highlighted in yellow):

```
override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath) as HotelTableViewCell

    // Configure the cell...
    let hotel = hotels[indexPath.row]
    cell.nameLabel.text = hotel.name
    cell.addressLabel.text = hotel.address
    cell.descriptionLabel.text = hotel.description

    // Set the font style
    cell.nameLabel.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline)
    cell.addressLabel.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleSubheadline)
    cell.descriptionLabel.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleBody)

    return cell
}
```

The *preferredFontForTextStyle* method of UIFont returns the system font with size based on the user's current content size for a specified text style. Here are the available text style options:

- UIFontTextStyleHeadline
- UIFontTextStyleSubheadline
- UIFontTextStyleBody
- UIFontTextStyleFootnote
- UIFontTextStyleCaption1
- UIFontTextStyleCaption2

Lastly, add the *deinit* method to remove the observer:

```
deinit {
    NSNotificationCenter.defaultCenter().removeObserver(self)
}
```

Now you can test the app again. When the app is launched in the simulator, press command+shift+h to go back to the home screen. Then go to Settings > General > Accessibility > Larger Text and enable the Larger Accessibility Sizes option. Drag the slider to set to change the text size.

Once changed, press command+shift+h again to go back to home screen. Launch the SelfSizingCell app and the size of the labels should change automatically.

USING CUSTOM FONT

Now that you have learned how to adopt Dynamic Type and enable your app to respond to text size changes. However, there is one issue with dynamic type that we haven't covered yet. The font of the text style is defaulted to Helvetica Neue. Fortunately, you can use *UIFontDescriptor* to get a font descriptor for a given text style. Based on the font descriptor, you can create your own font using UIFont. Here is an example:

```
var bodyFontDescriptor =
UIFontDescriptor.preferredFontDescriptorWithTextStyle(UIFontTextStyleBody)
let bodyFont = UIFont(name: "Avenir-Medium", size:
bodyFontDescriptor.pointSize)
cell.descriptionLabel.font = bodyFont
```

UIFontDescriptor provides a class method called *preferredFontDescriptorWithTextStyle* that returns a font descriptor of a given text style. In the sample, we ask for the font descriptor of body text style (UIFontTextStyleBody). The font descriptor contains the font size of the system font with respect to the current content size setting. With the font size, we can create our own font using UIFont and assign it to the cell label.

This solution works, but there is an even better way to apply your preferred fonts with Dynamic Type. Swift offers a feature called "extensions" that allows developers to add new functionality to an existing class. You can even extend the functionality of a built-in class (e.g. UIFont). If you have Objective-C background, extensions are similar to categories in Objective-C.

We will create an extension for UIFont and add a new method called *preferredCustomFontForTextStyle* to the class. To create an extension, you declare the extension with the *extension* keyword. Insert this extension in the HotelTableViewController.swift file and put it right after the "import UIKit" statement:

```
extension UIFont {
    class func preferredCustomFontForTextStyle (textStyle: NSString) ->
UIFont {
        let font =
UIFontDescriptor.preferredFontDescriptorWithTextStyle(textStyle as
String)
        let fontSize: CGFloat = font.pointSize
        if textStyle == UIFontTextStyleHeadline {
            return UIFont(name: "AvenirNext-Medium", size: fontSize)!
        } else if (textStyle == UIFontTextStyleSubheadline) {
            return UIFont(name: "Avenir-Medium", size: fontSize)!
        } else {
            return UIFont(name: "Avenir-Light", size: fontSize)!
        }
    }
}
```

The method simply returns a custom font based on the given text style. For instance, we used AvenirNext-Medium font for the headline text style.

You can use the method of the extension just like any other methods. Simply call up the *UIFont.preferredCustomFontForTextStyle* method instead of *UIFont.preferredFontForTextStyle*. So update the *tableView(_:cellForRowAtIndexPath:)* method like this:

```
override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
```


```
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath) as! HotelTableViewCell
    // Configure the cell...
    let hotel = hotels[indexPath.row]
    cell.nameLabel.text = hotel.name
    cell.addressLabel.text = hotel.address
    cell.descriptionLabel.text = hotel.description

    // Set the font style
    cell.nameLabel.font =
UIFont.preferredCustomFontForTextStyle(UIFontTextStyleHeadline)
    cell.addressLabel.font =
UIFont.preferredCustomFontForTextStyle(UIFontTextStyleSubheadline)
    cell.descriptionLabel.font =
UIFont.preferredCustomFontForTextStyle(UIFontTextStyleBody)

    return cell
}
```

Now compile and run the project again. The app should use the custom font instead of the system font. For your reference, you can download the final project from https://www.dropbox.com/s/q5qtvw1adk83jhv/SelfSizingCellDynamicType.zip?dl=0.