**Custom Cell**

So far, our cells have looked rather plain. Most applications use TableView to display its data but **sometimes it is hard to tell because they use Custom Cells**. Let's create a simple list with Custom Cells. Go ahead and create a new project called CustomCell with a Single Page Application template. Let's delete the ViewController that it is created for us and drag out a Table View Controller. **Make sure to set it as the Initial View Controller.**

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

Rename the ViewController.swift file to CustomCellsViewController.swift. Make sure that the CustomCellsViewController inherits from UITableViewController instead of UIViewController.

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

Now we can connect our Table View Controller to our code.

صورة تحتوي على نص

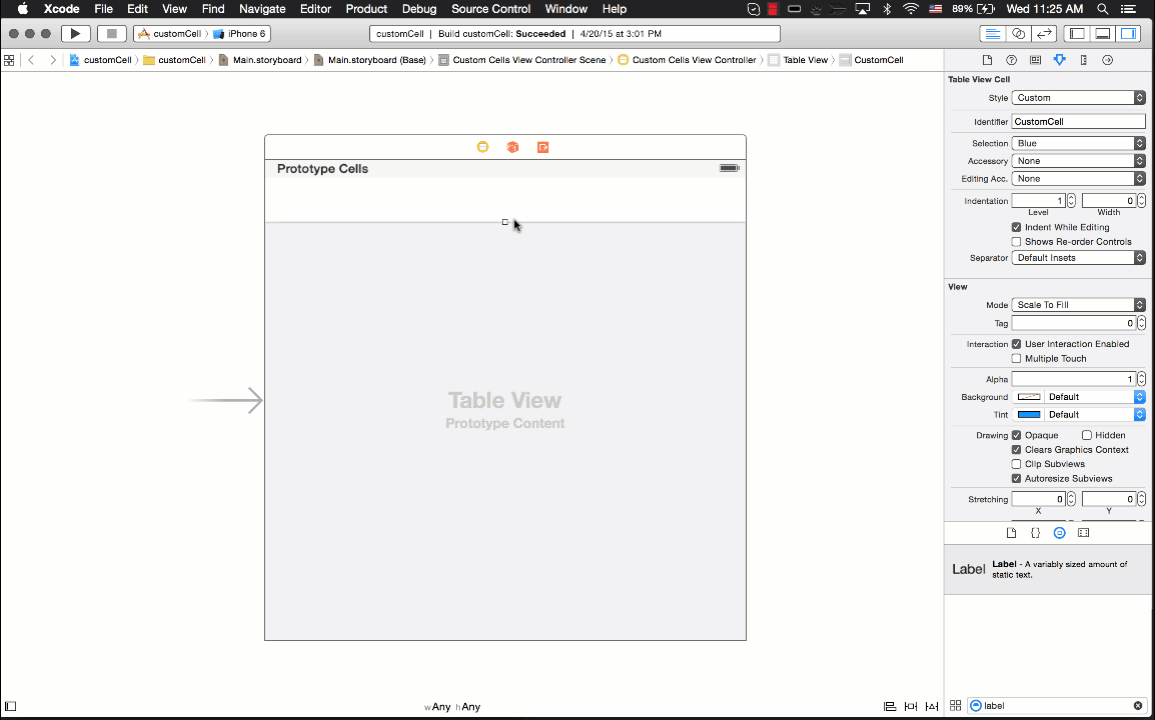
تم إنشاء الوصف تلقائياً

Then let's give our Prototype Cell an identifier to get rid of the warning.

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

Now we can start designing our Custom Cell.



We can't connect theButton and the Labels we have placed in the prototype cell because we don't know how many of them we are going to have. To set these attributes, we have to create a new class to represent this custom cell. Here we can make our Outlets and Actions. Create a new Swift file called CustomCell. Our class should look like the following for now:

// we have to use UIKit to have access to UITableViewCell

import UIKit

class CustomCell: UITableViewCell {

}

Now let's make an Outlet for the Label and the Button. However, we have to first tell Xcode that the prototype cell is going to be backed by the CustomCell class.

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

Now we can control drag to make the necessary connections:

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

We can now manipulate the attributes of this prototype cell which is an instance of the CustomCell class in our View Controller with the following code. We are going to just use an Array of Ints to serve as our Model object to demonstrate the use of this Custom Cell.

import UIKit

class CustomCellsViewController: UITableViewController {

var nums = [1, 90, 32, 23, 9, 12]

override func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

let cell = tableView.dequeueReusableCell(withIdentifier: "CustomCell") as! CustomCell

cell.rightLabel.text = "\(nums[indexPath.row])"

if nums[indexPath.row] > 24 {

cell.leftButton.backgroundColor = UIColor.green

} else {

cell.leftButton.backgroundColor = UIColor.red

}

// return cell so that Table View knows what to draw in each row

return cell

}

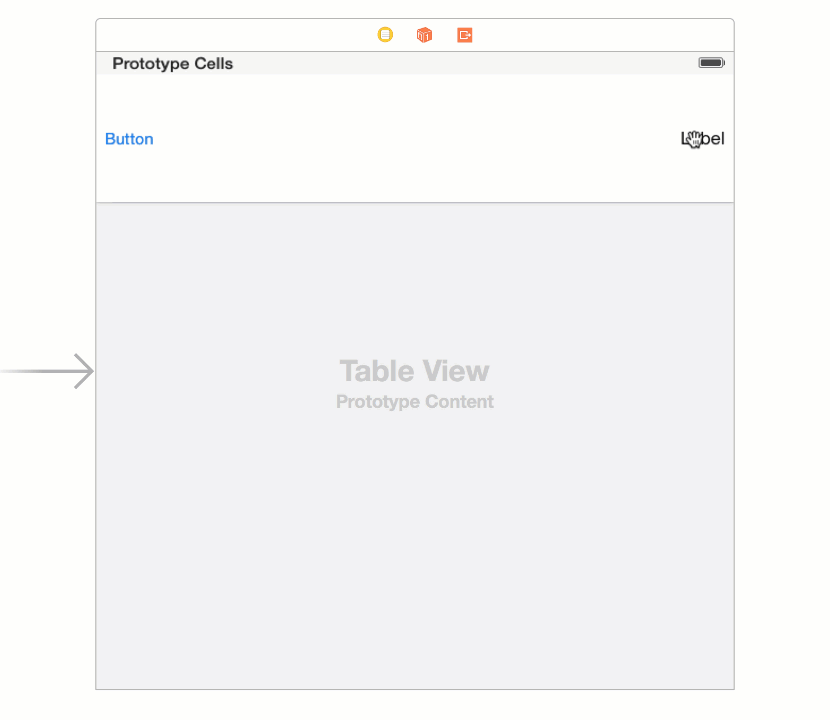
override func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

return nums.count

}

}

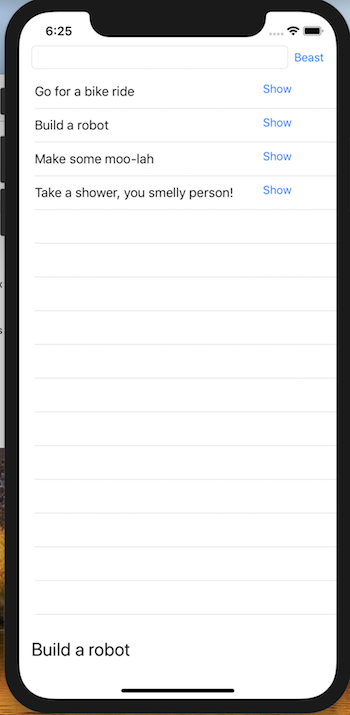
Wait...the buttons seem like they are working but where is the rightLabel? The rightLabel is there but it is outside of the screen. We have to use Auto Layout within the Cell so that the contents get shown properly. We can add Auto Layout by doing the following:



Congratulations! We have created our first Custom Cell together.

# Beast Cell Delegate & Protocol

Let's revisit our Beast List, and add a button on each row that will show the task description at the bottom of the screen.



First, let's go to the storyboard, adjust the table view by unpinning the bottom space from the view, and add a label at the bottom of the screen, and constrain appropriately. Be sure to create an IBOutlet for your new task description label in the View Controller. Next, we'll add a button to the prototype cell that says, "show".



Just like we did in the last chapter, we'll go ahead and **create a custom class** for the cell. Go to File -> New -> File and create a Cocoa Touch Class. Name your class something that makes sense, like BeastTableViewCell, and make sure it's a subclass of UITableViewCell.

In your class, **create an IBAction for the show button**.  In order for this to work, the class selected in the storyboard for the prototype must be set to your new class, BeastTableViewCell (the class drop-down menu under the newspaper icon).

import UIKit

class BeastTableViewCell: UITableViewCell {

@IBAction func showButtonPressed(\_ sender: UIButton) {

// this is where we should call a method to show the task in the main view

}

**But how can we change the label in the main View Controller class from the cell? This is where protocols and delegates come in!**

## Protocols and Delegates

A delegate is an object instance (like the main View Controller instance that gets created when we start the app) that does the work for another object instance (like a particular cell). In this example, the cell needs the view controller to change that bottom label to the task's description.

How will it know how to help the cell and do the work the cell needs? In order for the cell to communicate what it needs to happen, it will need a **protocol** **that any delegates will conform to**. A protocol tells the delegate what the cell will need it to do, without telling it how to do it. To create a protocol, create a new file (File -> New -> File) and select the generic Swift File type. We can call this file "BeastCellDelegate"

Put the following in your new file:

import Foundation

import UIKit

protocol BeastCellDelegate: class {

func showTaskDescription(description: String)

}

The cell will give the delegate (in this case, the view controller) a string, and then the view controller will have to decide what to do with that string. In order for the view controller to be the delegate, it has to state that it will conform to the protocol, and then, it has to implement anything outlined in the protocol.

import UIKit

class ViewController: UIViewController, BeastCellDelegate { // <- this is where it promises to conform to the protocol

var tasks = ["something", "something else", "final something"]

@IBOutlet weak var tableView: UITableView!

@IBOutlet weak var taskTextField: UITextField!

    // The new description we added at the bottom of the screen

@IBOutlet weak var taskDescriptionLabel: UILabel!

    // This is the new function the View Controller promised to implement!

func showTaskDescription(description: String) {

taskDescriptionLabel.text = description

}

Great! We now have a delegate! But how do these object instances communicate? If we make the View Controller a cell delegate, we can make a pointer to that delegate within the cell class so that when the "show" button is pressed within the cell, it can tell the delegate to go take care of the action.  Let's do that now.

import UIKit

class BeastTableViewCell: UITableViewCell {

    // Here we need to create a variable that will hold a reference to the delegate...

    var delegate: BeastCellDelegate?

@IBAction func showButtonPressed(\_ sender: UIButton) {

        // this is where we call the delegate's method to show the task in the task label

        delegate?.showTaskDescription(description: (self.textLabel?.text)!)

    }

Next we'll need to set the delegate to be the view controller instance, when the cell is being created. Remember how we create cells? That's right, cellForRowAt. In your view controller:

func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

        // Since we created a new class for the cells, we have to make sure the cell is the right type, by

        // casting it as a BeastTableViewCell

let cell = tableView.dequeueReusableCell(withIdentifier: "BeastCell", for: indexPath) as! BeastTableViewCell

cell.textLabel?.text = tasks[indexPath.row]

        // This line is very important! Now the cell has a reference to the view controller itself.

cell.delegate = self

return cell

}

# Show Segues + Navigation Controller

Arguably the most simple way to move from page to page is using a Navigation Controller and Show Segues.  We can lay out all of our views in a tree structure like so:

main

├── view a

│   └── view c

└── view b

├── view d

└── view e

To use this technique a view controller must be embedded within a navigation controller (In the storyboard, select a ViewController and click `Editor > Embed In > Navigation Controller`).  The main strength of the Navigation Controller is that it provides a button to dismiss the top view.

### Video Outline:

* Select main view controller, embed in navigation controller
* Create all children view controllers with buttons / titles
* Ctrl+drag from button on main view controller to child view controller
* Select "Show"