LECTURE #4

Welcome

Almost every application you use on iOS needs a way to display a list of data. Whether it be the Music app that displays all your songs, Settings to display all the ways you can customize your phone, or Game Center to show the leaderboard of your favorite game; all of these display this information using a table view. This lesson will teach the conceptual ideas behind table views and walk you through an example of displaying a set of data in a table.

Figure 4-1

MVC

iOS relies heavily on a concept known as *MVC*, which stands for *model-view-controller*. The idea is that your application should be developed in such a way that individual components should be separate from one another. Let's take the table view as an example (see Figure 4-1). What you see is a list of names presented in a **`UITableView`**. The names are being held by an array that we are storing in our *controller*, in most cases this is our view controller. What's important is that the data, otherwise known as the *model*, is separated from the *view* (the table view).

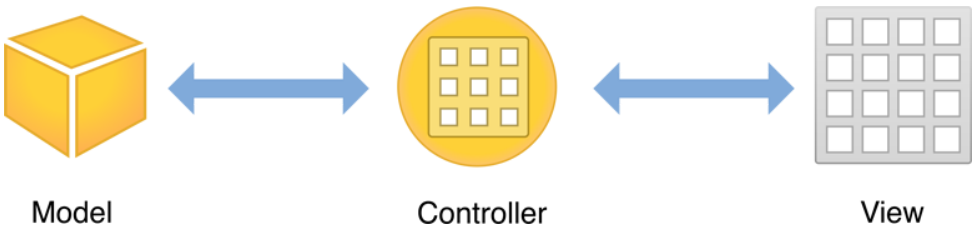


Figure 4-2

MVC is the cornerstone of iOS development. To quote from Apple:

> **Model-View-Controller (MVC) is central to a good design for any iOS app. MVC assigns the objects in an app to one of three roles: model, view, or controller. In this pattern, models keep track of your app's data, views display your user interface and make up the content of an app, and controllers manage your views. By responding to user actions and populating the views with content, controllers serve as a gateway for communication between models and views.**

That last sentence is one of the most important to keep in mind. For one last analogy, just imagine that the model and the view hate each other and they'll only communicate with each other via their mutual friend, the controller. We'll explore how this design pattern works in the case of table views.

Table Views

Go ahead and create a new view-based iOS project in Xcode. We're going to be developing the app shown in Figure 4-1. Head over to *Main.storyboard* and drag out a new *Table View* (not a Table View Controller) and drop it onto the main view. The view should now be filled with the table view and it should say "Table View Prototype Content". Before we add the content, control-drag from the table view to the view controller object and connect the **`dataSource`** and **`delegate`** properties of the table view. We also want to connect the **`UITableView`** to the **`ViewController`** class and call the property **`tableView`** (add it as a IBOutlet). Next, we want to add in a *Table View Cell* and drop it on top of our table view. You should now see a white strip running across the top of your tableview which is the prototype cell. We can customize this prototype cell inside interface builder to give it the look we want, but we'll talk about this later. With the newly added cell selected, select the Attributes Inspector and set the *Identifier* to "Cell". Lastly, embed the view controller inside a navigation controller from Xcode's Editor menu.

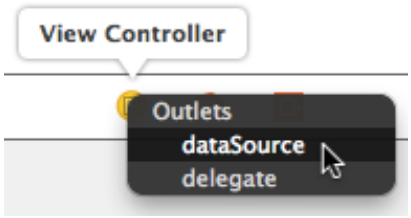


Figure 4-3

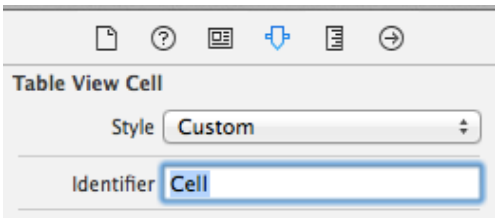


Figure 4-4

As we discussed in the section on *MVC*, we would like to keep the app's data separated from the views. In the case of the table view, we do not want to give direct access to the array of names that our controller holds, so the table view has to ask the controller for this information (remember, model no talky to view). What does a **`UITableView`** need to know? First off, it will ask our controller: "how many items do I need to display?" In our case, this is however many names are being kept in our array of names. So if we had 12 names, we'll respond to the table view telling it that we have 12 items. Once it knows how many names it has to display, then it can ask our controller: "what should I display at row _?" Our controller will then hand off the information at row 0, row 1, row 2, etc, until we've reached the number of elements we told the table view that we had. This concept of the view asking the controller for the information it needs to display is known as *delegation*. When we set our view controller to be the *dataSource* for our table view, this uses delegation to ask our view controller for the information it needs.

Moving on to *ViewController.swift*, let's configure our view controller to be the data source of our table view. To get started, our view controller should conform to the **`UITableViewDataSource`**:

```
class ViewController: UIViewController, UITableViewDataSource {
```

UITableViewDataSource is a protocol that defines methods that our view controller can implement so that the table view can display it's data. There are plenty of methods we can implement to configure the table view, but there are only 2 methods that are required. Let's add the following code to our class:

```
let names = ["Barney", "Andy", "Lucy", "Frank", "Marie"]

func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return names.count
}

func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:NSIndexPath)
     -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell") as UITableViewCell
    cell.textLabel.text = names[indexPath.row]
    return cell
}
```

The first thing we define is an array of names, which we call **names**. This is the model content that we want to display in the table view. Next we implement the 2 required methods in the **UITableViewDataSource** protocol, **tableView(_:numberOfRowsInSection:)** and **tableView(_:cellForRowAtIndexPath:)**. The first method is asking us to return the number of rows in a particular section. In our case we only have one section so we can simply return the number of names that we want to display (the number of rows). The second method we implement is asking for the cell at a particular row. In other words, it will ask what we should show at a particular row and we'll give back the view that it should display. We dequeue a cell from the tableview called "Cell". This is going to give us back the cell that we setup on our storyboard a little while ago (the cell that we gave the "Cell" identifier to). We get that back and infer that it is a **UITableViewCell** since we know that's the type of cell we added in Interface Builder. The **UITableViewCell** class has a property called **textLabel** which is the label built into the cell. To customize this cell, all we have to do is set the text to be the name at that particular row. Once we've setup our cell, we return it to the table view. That's it, build and run and check out your fully functional table view!

Let's talk about the counterpart to **UITableViewDataSource** which is the **UITableViewDelegate** protocol. While many of the methods in the data source protocol are meant for the tableview to receive it's data, the delegate is geared towards responding to events and altering the appearance on the table view (you're best to check out the documentation for both of these protocols to learn more about them). For example, we might want to know when the user taps a particular row, the table view will notify its delegate that a particular row was selected. We already set our view controller as the delegate in the storyboard so we're set there. Head over to the **ViewController** class and add the **UITableViewDelegate** as a protocol this class conforms to like this:

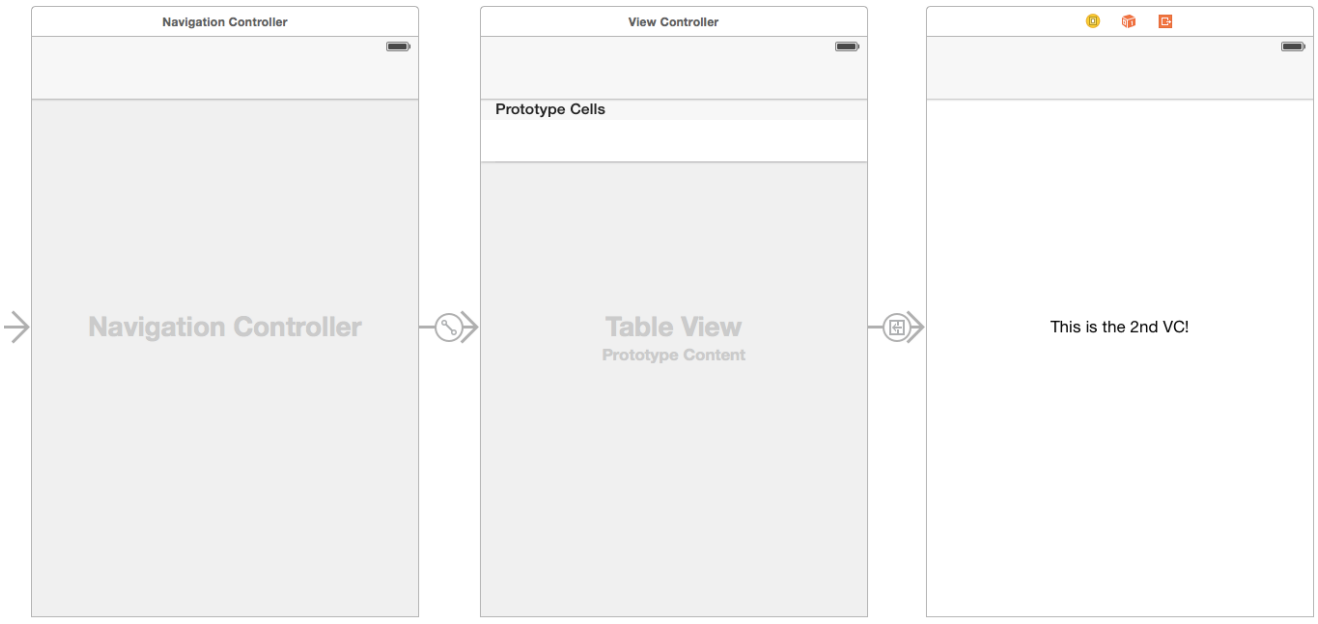
```
class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {
```

Let's try out a method for when we tap an a cell on our table view, go ahead and add the following to your **`ViewController`** class:

```
func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath)
{
    println("You selected row: \(indexPath.row)")
}
```

Now whenever we select a row on our table view, we'll be notified about which one we selected. You should see the output in the console each time you tap a cell on the table view.

For the last part of this lesson, we're going to demonstrate how we can select a cell and then transition to another view controller. It is common that the user would like to display more information about a particular item they tap and we can then transition them into a new view controller that displays more information. One way that we can make this a simple transition is by using a segue. Let's open up our Main.storyboard file again and add in a new view controller object to the scene and control-drag from the prototype cell in our table view to the new view controller. While you're at it, drag out a label and drop it on our new view controller, don't forget to rename the text with something fancy like "This is the 2nd VC!". Now that we've made a segue from the prototype cell to the next view controller, anytime a cell is selected, we will be transitioned to the next view controller.



(This is the 2nd part to Lecture 4) In the last bit we were limited to showing a single view controller with a tableview. Let's say now we want to be able to tap one of the names and display it in a new view controller, how can we pass this information on? First off, we'll have to create a new view controller class, so *File>New>File…*, select *Cocoa Touch Class*, and make a new subclass of **`UIViewController`** called **`DetailViewController`**. Once you've added it to your project, open the Main.storyboard file again. Add in a new view controller object to the scene and change it's class type to be "DetailViewController". Control-drag from the prototype cell in our table view to the new view controller and select the show segue, be sure to change its *Identifier* to be "DetailSegue" so we can reference it later. Activate the assistant editor mode (you should have the storyboard and the DetailViewController.swift visible, check back to Lesson 1 if you forget how this works). Add a new UILabel to our view controller's view in the storyboard and connect it to the DetailViewController class as an IBOutlet called **`label`**. Add a new String property to the DetailViewController class called **`name`**. The completed class should now look like this:

```
class DetailViewController: UIViewController {

    @IBOutlet weak var label: UILabel!
    var name = ""

    override func viewDidLoad() {
        super.viewDidLoad()
        label.text = name
    }
}
```

So why did we add the **`name`** property? From our **`ViewController`** class, we want to pass on whatever name is selected on the table view and pass that on to our **`DetailViewController`** when we transition via our segue. So let's implement this passing along of information. Head back to our **`ViewController`** class and add the following code to our class:

```
override func viewWillAppear(animated: Bool) {
    if let indexPath = tableView.indexPathForSelectedRow() {
        tableView.deselectRowAtIndexPath(indexPath, animated: false)
    }
}

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "DetailSegue" {
        let row = tableView.indexPathForSelectedRow()!.row
        let vc = segue.destinationViewController as DetailViewController
        vc.name = names[row]
    }
}
```

In **`viewWillAppear(_:)`**, we are deselecting the row if there is any selected. The reason we do this is because after we select a row and move onto the next view, we don't want that selection to remain on the table view. Feel free to try without implementing that method and see what happens. **`prepareForSegue(_:_:)`** is where we pass the data along to our **`DetailViewController`**. We first check to make sure we're using the "DetailSegue" which we created in our storyboard. Then we can

reference the **`name`** property on the **`DetailViewController`** and set it to the contents of the selected row.

That's all there is for this lesson, but there is a fair amount to digest. You should have a better understanding of how applications are designed using MVC, how we can apply that using datasources/delegates, and how table views work. Until next time.