# iOS: Multiple Selections in Table View

## Part 3. Continue adopting MVVM in iOS Development

Stan Ostrovskiy Follow
Jun 3, 2017 · 8 min read

This is the third Tutorial in my Table View with MVVM series. In the first two parts, we created the Table View with dynamic cells of different types, and we added a collapsible sections feature. Today, we will go through another common-used Table View scenario: multiple selections.

In many use cases, we need to create a Table View, that allows the user to select multiple cells. Usually, the API provides the list of items, that you display in Table View, and the user can select a few of this items to use them later (send to the backend, save to local storage, pass to the next screen, etc). Sometimes, you want to limit the selection by the certain number, or you don't want the user to proceed with no items selected.

Here is what I usually see in existing apps, or in some answers on *stackoverflow*:

```
let allItems = [Items]()
var selectedItems = [Items]()
```

Then, when the cell is selected, there is a what-the-hell-is-going-on-here way to add or remove the item at the selected index.

Another solution is little nicer: get the index of currently selected *TableView* rows, and then map it to the array of data source items:

```
if let selectedIndexes = tableView.indexPathsForSelectedRows {

   // tricky way to map indexes to the existing array of data
   // let selectedItems = ...
}
```

And while the second way feels more natural to the *TableView*, because it utilizes the built-in method *indexPathsForSelectedRows,* it's still not perfect if you want to separate your *ViewModel* from the *View*. Mapping the *tableView* indexes to the model items a good example of *Massive View Controller* pattern, that we want to avoid.

This becomes even messier if you want to keep the selections when you reload the *tableView:* you need to save the currently selected indexes, reload the *tableView*, and then call *selectRow* for each selected index. But what is the dataSource has changed between the reloads?

When you finish this tutorial, you will have the following code in your *TableViewController* to handle the multiple row selection:

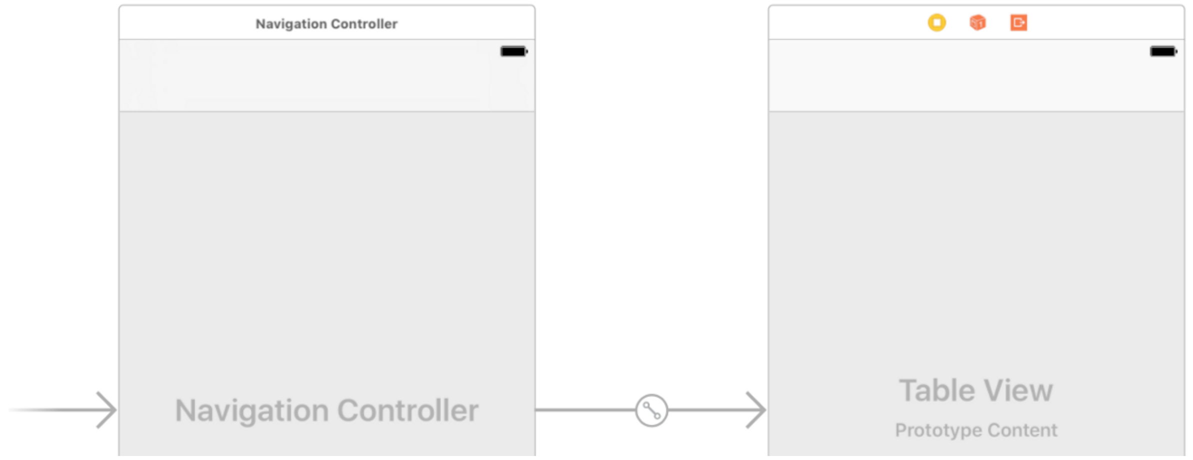
```
// what code?
```

That’s right. No code at all. Everything will be handled by *ViewModel* and the the *TableViewDelegate*. And the selections will stay persistent when you reload the whole *TableView* or any of its sections.

Sounds good? Let’s start!

. . .

We begin with building the UI. First, create the *UIViewController*, embed in in *NavigationController*, and add the *tableView* and the *NextButton* to it:



View Controller Setup

> *Do not set tableViewDataSource and TableViewDelegate in storyboard at this point. We will do it later in the code.*

Create the *tableView* and *nextButton* outlets and *next* action in *ViewController* class:

```
class ViewController: UIViewController {
   @IBOutlet weak var tableView: UITableView?
   @IBOutlet weak var nextButton: UIButton?

   @IBAction func next(_ sender: Any) {

   }
}
```

Add one custom cell with *titleLabel:*

```
class CustomCell: UITableViewCell {
   @IBOutlet weak var titleLabel: UILabel?

   override func awakeFromNib() {
      super.awakeFromNib()
      selectionStyle = .none
   }
}
```

Set *selectionStyle* to *none*, because we will use the *Checkmark AccessoryView* to indicate selection.

. . .

Now we can start the most interesting part. We create a simple *Model*:

```
struct Model {
   var title: String
}
```

We only have the title in this *Model*, but in the real-world apps, it can have a more complex structure with nested objects, that you parse from JSON. The model will represent the item titles, that we display in the *TableView*. But it does not know if the item is selected or not, and it should not now it. This is when we need to bring the *ViewModel* in place.

In our case, the *ViewModelItem* will own the model, and it will also hold the item title and the current selection state:

```
class ViewModelItem {
   private var item: Model

   var isSelected = false
   var title: String {
      return item.title
   }

   init(item: Model) {
      self.item = item
   }
}
```

As you see, *ViewModelItem* holds the variables, that we need to use in the *TableView*. For example, if we have a custom cell with description label and the picture, we would add the appropriate variables to the *ViewModelItem*.

> *Note, that the Model variable is private. In this case, only the ViewModel can access it, and this is what we want to achieve in MVVM. View knows nothing*

> *about the Model.*

Having the *ViewModelItem* we can create the *ViewModel*:

```
class ViewModel {
   var items = [ViewModelItem]()
}
```

Here we will have another simplification and create a static data array to initialize the ViewModel. We pretend we received this data from the backend and parsed it to the array of *Model* items:

```
let dataArray = [Model(title: “Swift”),
    Model(title: “Objective C”),
    Model(title: “Java”),
    Model(title: “Kotlin”),
    Model(title: “Java Script”),
    Model(title: “Python”),
    Model(title: “Ruby”),
    Model(title: “PHP”),
    Model(title: “Perl”),
    Model(title: “Go”),
    Model(title: “C#”),
    Model(title: “C++”),
    Model(title: “Visual Basic”),
    Model(title: “Pascal”)]
```

To initialize a *ViewModel* we simply map this array to the array of *ViewModelItems*:

```
class ViewModel {
   var items = [ViewModelItem]()
   init() {
      items = dataArray.map { ViewModelItem(item: $0) }
   }
}
```

. . .

For now, switch to the *CustomCell* class and add an Item, that we will use to configure the cell:

```
var item: ViewModelItem?
```

Using the property observer, and connect the item title property to the titleLabel:

```
var item: ViewModelItem ? {
   didSet {
      titleLabel?.text = item?.title
   }
}
```

We also need to override *setSelected* method:

```
override func setSelected(_ selected: Bool, animated: Bool) {
   super.setSelected(selected, animated: animated)

   // update UI
   accessoryType = selected ? .checkmark : .none
}
```

This will take care of the UI updates for selection states: when the cell is selected, we display the *Checkmark*.

When the cell is selected, we also need to updated the *ViewModel*. Add a *tableViewDelegate* to the *ViewController*:

```
extension ViewController: UITableViewDelegate {
   func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
      viewModel.items[indexPath.row].isSelected = true
   }

func tableView(_ tableView: UITableView, didDeselectRowAt indexPath:
IndexPath) {
      viewModel.items[indexPath.row].isSelected = false
   }
}
```

When the cell is selected/deselected, it will trigger the appropriate delegate method and update the current *viewModelItem*.

Return to the *ViewController* and create and initialize the *ViewModel* property:

```
class ViewController: UIViewController {

   var viewModel = ViewModel()

   @IBOutlet weak var tableView: UITableView?
   @IBOutlet weak var nextButton: UIButton?
}
```

Do a basic *tableView* setup in *ViewDidLoad:* register cell and provide the cell height.

To allow multiple selections, add one more line of code:

```
tableView?.allowsMultipleSelection = true
```

Following the MVVC structure, we will not use a *ViewController* as a tableView *dataSource*. Instead, our *ViewModel* will be a *dataSource*:

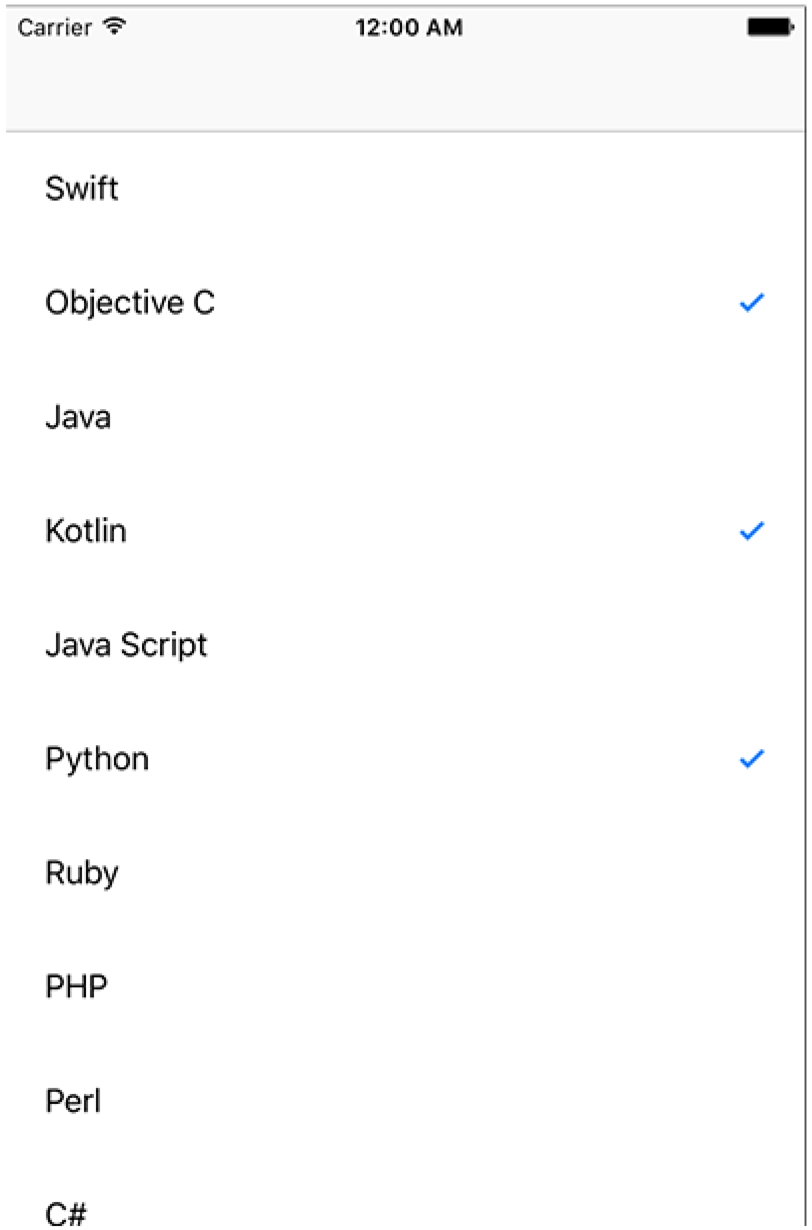
```
tableView?.dataSource = viewModel

// set delegate to self
tableView?.delegate = self
```

To get rid of the compiler error, add an extension to *ViewModel*:

```
extension ViewModel: UITableViewDataSource {
   func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
      return items.count  // (1)
   }


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


      if let cell = tableView.dequeueReusableCell(withIdentifier:
CustomCell.identifier, for: indexPath) as? CustomCell {
         cell.item = items[indexPath.row] // (2)


         // select/deselect the cell
         if items[indexPath.row].isSelected {
            tableView.selectRow(at: indexPath, animated: false,
scrollPosition: .none) // (3)
         } else {
            tableView.deselectRow(at: indexPath, animated: false) //
(4)
         }
         return cell
      }
      return UITableViewCell()
   }
}
```

For *numberOfRowsInSection,* we use the items count (1). We also dequeue our custom cell in *cellForRowAt* and set its item to the *ViewModelItem* at the current index (2). We also select/deselect the cell based on the *viewModelItem* state (3 and 4).

If you build and run the project now, we will be able to select and deselect multiple cells.



To make sure the selections are persistent if you reload the *TableView,* try to reload data when you tap next:

```
@IBAction func next(_ sender: Any) {

   // TEMP: check the persistent selections
   tableView.reloadData()
}
```

The only question is how to get the selected items when we need them?

We don't want to do any of this logic in our *View* (MVVM, remember?). So return to the *ViewModel* class and add a computed property that will take care of selected items using the *filter higher order* function:

```
class ViewModel {
   var items = [ViewModelItem]()
```

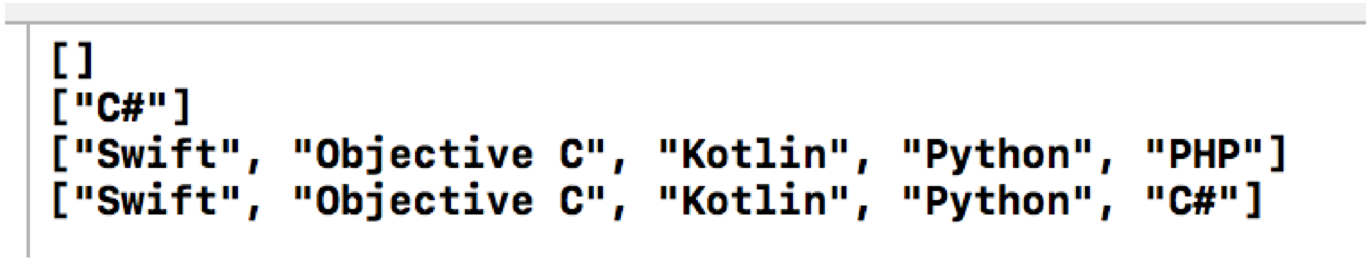
```
    var selectedItems: [ViewModelItem] {
        return items.filter { return $0.isSelected }
    }


    // ...

}
```

To test that all this setup work properly, we can add a print statement inside the *ViewControllers'* next action:

```
@IBAction func next(_ sender: Any) {
    print(viewModel.selectedItems.map { $0.title })
}
```

> *I am using map higher order function to convert the array of ViewModelItems to the array of Strings. If you want to get the String representation of your class, there a better way of doing that using CustomStringConvertable protocol. But I will stick to the map to keep the things simple.*

Build and run the project again, select some of the items, and tap “Next”. In the console log you will see the print of the currently selected items:



```
[]
["C#"]
["Swift", "Objective C", "Kotlin", "Python", "PHP"]
["Swift", "Objective C", "Kotlin", "Python", "C#"]
```

Array of selected items

Having these items, you can do whatever you need in your app: post to the backend, save, pass to the next screen, etc.

As you can see, it works with a very little code in the *ViewController*, keeping the business logic outside of the *View*.

. . .

Let’s do a few nice-to-have additions:

- limit the number of selected items by 3
- don’t allow the user to tap “Next” if no items are selected

Good news: both tasks sound much more difficult than they are.

For the first one, we need to use *willSelectRowAt* delegate method:

```
func tableView(_ tableView: UITableView, willSelectRowAt indexPath:
IndexPath) -> IndexPath? {
   if viewModel.selectedItems.count > 2 {
      return nil
   }
   return indexPath
}
```

*WillSelectRowAt* will fire when the user taps the cell and before it gets selected. So we intercept it with a simple check: if the currently selected items count is greater than 2, we don't allow this cell to be selected (return nil). Otherwise, allow the selection by returning the selected *IndexPath*.

Run the project and test it. Now you will not be able to select more than 3 items.

To disable the *nextButton* when there is no selection, we need to detect when the cell selection is changed.

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
   // update ViewModel item
   viewModel.items[indexPath.row].isSelected = true

   nextButton?.isEnabled = !viewModel.selectedItems.isEmpty
}

func tableView(_ tableView: UITableView, didDeselectRowAt indexPath:
IndexPath) {
   // update ViewModel item
   viewModel.items[indexPath.row].isSelected = false

   nextButton?.isEnabled = !viewModel.selectedItems.isEmpty
}
```

We also need to check the selection in *ViewDidLoad* (in case if some items should be selected by default):

```
nextButton?.isEnabled = !viewModel.selectedItems.isEmpty
```

That's all! Try it again, and it will disable the nextButton if no items are selected.

. . .

You can find the final project here:

**Stan-Ost/TableViewWithMultipleSelection**

Contribute to TableViewWithMultipleSelection development by creating an account on GitHub.

github.com

Thanks for reading! Questions or comments — feel free to ask.

iOS iOS App Development Mobile App Development Tutorial Swift