

Advanced Mobile Application Development

Week 4: Navigation Controllers

Navigation Controllers

Navigation controllers manage transitions forwards and backwards through a series of view controllers.

<https://developer.apple.com/documentation/uikit/uINavigationController>

- provides a drill-down interface for hierarchical data
- often used with table views

On the smaller iPhones hierarchical data is best shown using a succession of table views.

On the iPhone Plus and iPad a split view is used.

The **UINavigationController** class has two main components

- Stack of controllers
 - The navigation controller provides a container for the stack of controllers
 - The initial view controller on the stack is the root view controller
 - Subsequent view controllers are subcontrollers or child controllers
- Navigation bar

Navigation Bars

The **UINavigationController** class enables the navigation bar to manage navigation between different views in a navigation controller <https://developer.apple.com/documentation/uikit/uINavigationController>

<https://developer.apple.com/design/human-interface-guidelines/ios/bars/navigation-bars/>

Navigation bar buttons (defaults)

- Left: Back button often labeled with the title of the view it takes you back to
- Middle: Navigation bar title
 - Use a large title when you need to provide extra emphasis on context
- Right: Empty; edit or done button for managing content in the current view

You can also customize the appearance of a navigation bar.

iOS11 added the ability to have large titles like you see in Mail, Phone, or Settings.

But large titles take a lot of extra screen real estate so don't over use them.

When you do use it as the user scrolls it will automatically shrink so it doesn't take up a lot of room.

Large title use:

- Use purposefully
- Wayfinding - make it very obvious to the user where they are in the app
- Top level of navigation
 - Mail uses large titles on the first two screens and once the user drills down into a specific email the titles are standard sized
- Distinguish between similar views (mail)
 - Clock app has views with distinct layouts and no data to sort through so large titles would just compete with the content

To use large titles you have to set the **UINavigationController** **prefersLargeTitles** property to true (default is false). <https://developer.apple.com/documentation/uikit/uINavigationController/2908999-preferslargetitles>

Then you can use the **UINavigationControllerItem** **largeTitleDisplayMode** property to configure the title's appearance so you can control at each level if the large size or the standard size title is used.

<https://developer.apple.com/documentation/uikit/uINavigationControllerItem.largeTitledisplaymode>

Segues

When configuring segues between view controllers on a navigation stack use the Show and Show Detail segues:

- Show segue—The navigation controller pushes the specified view controller onto its navigation stack.
- Show Detail segue—The navigation controller presents the specified view controller modally such as in a split view controller on the iPad and plus size phones.

Table View Cell Accessories

Table view cells can include accessories that help indicate to the user what can be done by that cell.

<https://developer.apple.com/documentation/uikit/uitableviewcellaccessorytype>

- Disclosure indicator: chevron image used when selecting a cell results in the display of another view reflecting the next level in the data-model hierarchy. (mail)
- Detail button: info button reveals additional details or functionality related the item (phone)
- Detail disclosure button: chevron and info buttons are used when there are 2 different options for that row: one action when the user taps the row and another when they tap the detail button.
- Checkmark when a touch on a row results in the selection of that item. This kind of table view is known as a selection list, and it is analogous to a pop-up list. Selection lists can limit selections to one row, or they can allow multiple rows with checkmarks.

Table View Cell Types

Dynamic prototype cells let you design one cell and use it as a template for other cells in the table.

Static cells enable you to design a table with a fixed number of rows

- use when you know what the table looks like at design time
- these are static in the sense that they will exist every time the app is run
- less code than dynamic prototype cells
- the content of static cells can still change but the number does not (ex: Settings)

Countries

File | New Project

iOS App

Product Name: countries

Team: None

Org identifier: ATLAS (can be anything, will be used in the bundle identifier)

Interface: Storyboard

Language: Swift

Uncheck core data and include tests.

Uncheck create local git repo

Root view controller

Go into MainStoryboard. The initial scene is a view controller but we want a table view controller. Click on the scene and delete it. Then drag onto the canvas a table view controller.

Instead of modifying ViewController.swift I'm going to create a new class to control this view.

File | New | File

iOS | Cocoa touch class

ContinentTableViewController subclass of UITableViewController

Now go back into MainStoryboard and select the view controller and change its class to ContinentTableViewController.

In the document hierarchy select the table view and in the Connections inspector check that the dataSource and delegate for the table view are set to ContinentTableViewController. If not, drag from the circles to the View Controller icon.

In the document hierarchy select the Table View Cell and in the attributes inspector make the identifier “continentCell”.

Now for this table view controller to be controlled by a navigation controller, with the controller selected go to Editor | Embed in | Navigation Controller (or use the Embed In icon).

This creates a navigation controller and assigns the initial scene as the root view controller.

With the Navigation Controller selected go into the attributes inspector and check Is Initial View Controller.

A relationship segue from the navigation controller to the table view controller was also created.

Select the Navigation Item that was added to the table view controller and make the title Continents. Large title should be Automatic.

Drag continents2.plist into your app and make sure you have Copy items if needed checked.

It’s an Array of dictionaries that have a key “continent” with the value of type String, and a key “countries” with the value an array of type String.

Next we’re going to create a struct for our data model.

File | New File

iOS Source Swift File

ContinentsData

Make sure it’s saving to your project folder and the target is checked.

```
struct ContinentsData : Codable {  
    var continent : String  
    var countries : [String]  
}
```

Now we’ll add a class to load and control our data. This is similar to our last app but we also include a method to add and delete countries.

File | New File

iOS Source Swift File

ContinentsDataLoader

Make sure it’s saving to your project folder and the target is checked.

```
class ContinentsDataLoader{  
    var allData = [ContinentsData]()  
  
    func loadData(filename: String){  
        if let pathURL = Bundle.main.url(forResource: filename,  
withExtension: "plist"){  
            //creates a property list decoder object  
            let plistdecoder = PropertyListDecoder()  
            do {  
                let data = try Data(contentsOf: pathURL)
```

```

        //decodes the property list
        allData = try plistdecoder.decode([ContinentsData].self,
from: data)
    } catch {
        // handle error
        print(error)
    }
}

func getContinents() -> [String]{
    var continents = [String]()
    for item in allData{
        continents.append(item.continent)
    }
    return continents
}

func getCountries(index:Int) -> [String] {
    return allData[index].countries
}

func addCountry(index:Int, newCountry:String, newIndex: Int){
    allData[index].countries.insert(newCountry, at: newIndex)
}

func deleteCountry(index:Int, countryIndex: Int){
    allData[index].countries.remove(at: countryIndex)
}
}

```

In ContinentTableViewController.swift create an array of Strings for the list of continents, instance of the ContinentsDataLoader class to load and access our data, and a constant for our file name.

```

var continentList = [String]()
var continentsData = ContinentsDataLoader()
let dataFile = "continents2"

```

Let's load the data and get the continent list in viewDidLoad(). We do it here since the list of continents never changes.

```

override func viewDidLoad() {
    super.viewDidLoad()
    continentsData.loadData(filename: dataFile)
    continentList=continentsData.getContinents()
}

```

Now let's get our table set up. This process will be similar to our last app. Either update numberOfSections(in:) to return 1 or delete it.

```

override func numberOfSections(in tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 1
}

```

```
}
```

Implement the two required methods for the UITableViewDataSource protocol.

```
//Number of rows in the section
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
    return continentList.count
}
```

Uncomment and implement this method

```
// Displays table view cells
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "continentCell", for: indexPath)
    var cellConfig = cell.defaultContentConfiguration()
    cellConfig.text = continentList[indexPath.row]
    cell.contentConfiguration = cellConfig
    return cell
}
```

Run it and you should see the continent data listed in the table view. Although this looks the same as last time, we're going to be able to navigate from the cells to a detail view controller which we'll add next.

To enable large titles, add to viewDidLoad()

```
//enables large titles
navigationController?.navigationBar.prefersLargeTitles = true
```

In the storyboard as long as the navigation item has the Large Title attribute set to Automatic or Always the title will be large. (this can also be set programmatically)

Detail view controller

Now we want to be able to select a continent and see its list of countries.

Go into MainStoryboard and drag a table view controller onto the canvas to the right to be the detail view controller.

Now we need a class to control it.

File | New | File

iOS | Cocoa touch class

CountryTableViewController subclass of UITableViewController

Once created double check that CountryTableViewController is a subclass of UITableViewController.

Also look to see that it's added methods stubs for all the table view methods we'll need.

Go back into the storyboard and make CountryTableViewController the class for your new table view controller in the identity inspector.

Select the table view cell in the country scene table view and in the attributes inspector give it the identifier countryCell.

Now let's make the segue from the root view controller to the new detail view controller.

Cntrl-click and drag from the root prototype cell to the detail view controller.

When you unclick you will get a popup and must choose a Selection Segue - Show.

Select your new segue and in the attributes inspector give it the identifier countrysegue.

In the root view controller select the table view cell and in the attributes inspector change accessory to disclosure indicator. This indicates that selecting that row will show related data. We're not going to set the title for this view in the storyboard but we're going to do it programmatically so it will say whatever continents' countries we're looking at. If you don't want large titles change Large Title to Never.

If you run it at this point the controller will navigate, we just have to load the data. Note how the back navigation automatically shows the title of the controller where the back button will take you.

In CountryTableViewController.swift create an instance of the ContinentsDataLoader class, a variable to hold the selected continent, and an array for the list of countries.

```
var continentsData = ContinentsDataLoader()
var selectedContinent = 0
var countryList = [String]()
```

Now we need to set up the countries for the selected continent. We're going to do this in viewWillAppear instead of viewDidLoad because we need to do this every time the view appears. viewDidLoad will only be called the first time the view is loaded.

```
override func viewWillAppear(_ animated: Bool) {
    countryList = continentsData.getCountries(index: selectedContinent)
}
```

Now let's update the delegate protocol methods. These should look familiar.

```
override func numberOfSections(in tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 1
}
```

```
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
    // #warning Incomplete implementation, return the number of rows
    return countryList.count
}
```

Uncomment and implement this method

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "countryCell", for: indexPath)
    var cellConfig = cell.defaultContentConfiguration()
    cellConfig.text = countryList[indexPath.row]
    cell.contentConfiguration = cellConfig
    return cell
}
```

The last part is for the root view controller to set data in the detail view controller. Go into ContinentTableViewController.swift

We need it to tell the detail view controller which continent was selected and we'll also set the title to the name of the continent and set the ContinentsDataLoader instance. The prepare(for: sender:) method is called when a segue is about to be performed so it's a good place to configure the destination view controller.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "countrysegue" {
        if let countryVC = segue.destination as?
CountryTableViewController {
            if let indexPath = tableView.indexPath(for: (sender as?
UITableViewCell)!) {
                //sets the data for the destination controller
                countryVC.title = continentList[indexPath.row]
                countryVC.continentsData = continentsData
                countryVC.selectedContinent = indexPath.row
            }
        }
    }
}
```

Make sure the segue identifier exactly matches the one you used in the storyboard.

The segue destination property is of type UIViewController. In order to work with the CountryTableViewController class we need to down cast the object returned to be of type CountryTableViewController which is a subclass of UIViewController.

We use as?, the optional type cast operator, for the downcast which will return nil if it's not possible. If it succeeds then the CountryTableViewController instance is assigned to the countryVC constant and goes into the body of the if statement.

We do the same thing for the indexPath.

Now we can pass the data by assigning it to the properties in the CountryTableViewController class.

If the detail view controller has large titles and you don't want them you can set the Navigation Item's Large Title property in the storyboard (or code) to never.

If you run it now when you select a continent it will segue and show you the countries listed for that continent and the title will have the continents name.

Delete rows

Now let's add the ability to delete countries.

In CountryTableViewController.swift in viewDidLoad uncomment the following line

```
self.navigationItem.rightBarButtonItem = self.editButtonItem
```

(although you can add a bar button item called Edit in the storyboard, it will not automatically call the methods needed, so you should do it programmatically.)

Uncomment the following:

```
override func tableView(_ tableView: UITableView, canEditRowAt
indexPath: IndexPath) -> Bool {
    // Return false if you do not want the specified item to be
    editable.
    return true
}
```

Uncomment and implement

```
    override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellStyleEditingStyle, numberOfRowsInSection indexPath: IndexPath) {
        if editingStyle == .delete {
            //Delete the country from the array
            countryList.remove(at: indexPath.row)
            // Delete the row from the table
            tableView.deleteRows(at: [indexPath], with: .fade)
            //Delete from the data model instance
            continentsData.deleteCountry(index: selectedContinent,
countryIndex: indexPath.row)
        } else if editingStyle == .insert {
            // Create a new instance of the appropriate class, insert it into the array, and add a new
row to the table view
        }
    }
```

Once editing is allowed you can use the Edit button to delete a country or swipe to delete. Instead of deleting the row from the table we could just reload the table after we remove it from the array.

```
tableView.reloadData()
```

Move Rows

Now let's make the rows moveable. In CountryTableViewController find the stub for this method and un-comment it.

```
    override func tableView(_ tableView: UITableView, canMoveRowAt
indexPath: IndexPath) -> Bool {
        // Return false if you do not want the item to be re-orderable.
        return true
    }
```

Now un-comment and implement the movement of the rows.

```
    override func tableView(_ tableView: UITableView, moveRowAt
fromIndexPath: IndexPath, to: IndexPath) {
        let fromRow = fromIndexPath.row //row being moved from
        let toRow = to.row //row being moved to
        let moveCountry = countryList[fromRow] //country being moved
        //swap positions in array
        countryList.swapAt(fromRow, toRow)
        //move in data model instance
        continentsData.deleteCountry(index: selectedContinent, countryIndex:
fromRow)
        continentsData.addCountry(index: selectedContinent, newCountry:
moveCountry, newIndex: toRow)
    }
```

To move rows in the simulator click on the Edit button and then grab the 3 lines to move a row.

Add countries

Let's add the ability to add countries.

Go into Main.storyboard and add a new view controller.

Add a label and a textfield to the view so the user can enter a new country.

Create a new Cocoa Touch class to be its controller called `AddCountryViewController`, subclass of `UIViewController`.

Back in the storyboard make the `AddCountryViewController` class the controller for the new view controller.

Now let's make an outlet connection for the textfield called `countryTextfield`. Remember you must make this connection to `AddCountryViewController.swift`.

Go into `CountryTableViewController.swift` and in `viewDidLoad()` comment out

```
self.navigationItem.rightBarButtonItem=self.editButtonItem()
```

Back in the storyboard go into the Country Table View Controller scene and add a bar button item to the navigation item and change its System Item to Add.

Create a segue from the add button in the Country Table scene to the Add Country scene and choose a Present Modally segue. Use the document hierarchy to ensure you have the bar button selected for the connection.

We don't want push navigation because that's designed for a drill-down interface, where you're providing more information about whatever the user selected. Adding an item is a modal operation—the user performs some action that's complete and self-contained, and then returns from that scene to the main navigation.

If you run it now the add button will bring up the Add Country scene, but you'll notice there's no way for it to go back. Because a modal view controller doesn't get added to the navigation stack, it doesn't get a navigation bar from the table view controller's navigation controller. However, you want to keep the navigation bar to provide the user with visual continuity.

Using the object library add a navigation bar to the top of the Add Country scene and change the navigation item title to Add New Country.

Add two bar button items. Put one on the right and change it to Done, and the other on the left and change it to Cancel.

You can also change the bar tint to the system background color so it matches the nav bar in the detail controller.

You'll also need to fix the constraints for the Add Country scene.

The navigation bar will need constraints – leading, trailing, and top. You can put the label and textfield in a horizontal stack view and position its top to have vertical spacing from the nav bar and leading and trailing constraints to the safe area or superview.

Make sure you're adding these constraints to the stack view and not the label or textfield.

If you run it at this point you should see the nav bar with the buttons but they don't do anything yet. Let's implement the Cancel button by setting up an unwind segue to undo the modal segue back to `CountryTableViewController`.

First we have to create an unwind method in the destination view controller, `CountryTableViewController`. A method that can be unwound to must return an `IBAction` and take in a `UIStoryboardSegue` as a parameter.

```
@IBAction func unwindSegue (_ segue:UIStoryboardSegue){ }
```

We will implement it later.

Now go back into the storyboard and from the cancel button in the Add Country scene, control click and connect to the Exit icon in the dock and choose this unwind method.

Choose this segue and give it the identifier cancelSegue.

Do the same thing for the Done button and give it the identifier doneSegue.

Since both will call the same method we'll distinguish between the two segues by using the identifier when we implement the unwind method.

If you run it now the done and cancel buttons will both take you back to the detail view controller, but your data is not saved yet.

Add Data

Go into AddCountryViewController.swift

Define a variable to store the new country.

```
var addedCountry = String()
```

When the user taps Done we want to get the value from the textfield.

The prepare(for: sender:) method is called when a segue is about to be performed so it's a good place to save data.

If the user leaves the textfield empty and taps Done an empty row will be added. You can tell it adds an empty row because if you swipe that row you get the delete option (which you don't get on a row that doesn't exist). We use .isEmpty to test to avoid adding empty rows.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "doneSegue" {
        //only add a country if there is text in the textfield
        if countryTextfield.text?.isEmpty == false {
            addedCountry = countryTextfield.text!
        }
    }
}
```

In CountryTableViewController we implement the unwind method

```
@IBAction func unwindSegue(_ segue: UIStoryboardSegue) {
    if segue.identifier == "doneSegue" {
        if let source = segue.source as? AddCountryViewController {
            //only add a country if there is text in the textfield
            if source.addedCountry.isEmpty == false {
                //add country to our data model instance
                continentsData.addCountry(index: selectedContinent,
newCountry: source.addedCountry, newIndex: countryList.count)
                //add country to the array
                countryList.append(source.addedCountry)
                tableView.reloadData()
            }
        }
    }
}
```

We don't save the data if the cancelSegue is triggered. We could also connect the Cancel button to its own method that dismisses the view controller.

Run your app and you should be able to add new countries. Try to add an empty row. When you swipe you don't get the Delete icon which means it wasn't added.

Continent Info

Now let's add another table view this time using static cells.

Drag a table view controller onto the storyboard. Create a new Cocoa Touch class for it called ContinentInfoTableViewController. Make sure you make its superclass UITableViewController. Back in the storyboard set the class for the new scene to be ContinentInfoTableViewController.

Create a segue from the continents scene (root) cell to the new country info view controller and choose Accessory Action Show segue. Give it the identifier continentsegue.

In the continents scene change the accessory to a detail disclosure in the table view cell. This is to indicate that selecting a row will still drill down to the countries, but tapping on the detail accessory will bring us to our new table view that will have more info. Now when you run it tapping the detail disclosure indicator brings you to the new ContinentInfoTableViewController.

Static cells

Now let's get our static cells set up in the continent info scene.

In the storyboard select the table view in the new continent info scene and in the Attributes Inspector change Content to Static Cells and Style to Grouped.

Select the Table View Section (document hierarchy) and change Rows to 2 and header to Continent Info.

Select the first cell and use the attributes inspector to set its Style to Right Detail. Double-click to select the text of the label on the left and change it to Continent. Repeat the same steps for the second cell, changing its text to Number of countries.

Open the assistant editor so we can create outlets for the detail labels called continentName and countryNumber. Make sure you're making the connections to ContinentInfoTableViewController.swift and that you are making them from the detail labels (use the document hierarchy to check).

Go into ContinentInfoViewController.swift and add two variables

```
var name = String()
var number = String()
```

Delete the 2 dataSource methods as static cells don't use them.

Now if you run it you can see your static cells.

Now let's populate them with data that will be passed in.

```
override func viewWillAppear(_ animated: Bool) {
    continentName.text=name
    countryNumber.text=number
}
```

In ContinentTableViewController.swift we have to update prepare(for: sender:) to work with the detail disclosure accessory button by adding the following after the if statement:

```

//for detail disclosure
else if segue.identifier == "continentsegue"{
    if let infoVC=segue.destination as? ContinentInfoTableViewController {
        if let editingCell = sender as? UITableViewCell {
            let indexPath = tableView.indexPath(for: editingCell)
            infoVC.name = continentList[indexPath!.row]
            let countryList = continentsData.getCountries(index:
(indexPath?.row)!)
            infoVC.number = String(countryList.count)
        }
    }
}

```

The navigation bar might look like its height is too large. That's because it's inheriting the value for large titles and it doesn't even have a title. You can turn off large titles in ContinentInfoViewController in the storyboard by selecting the navigation item and making Large Title to Never.

Or you can do it programmatically by updating viewDidLoad()
`navigationItem.largeTitleDisplayMode = .never`

Data Persistence (countries data)

Let's add data persistence to our countries app.

In the ContinentsDataLoader class add a constant for the filename we'll write to.

```
let datafilename = "data.plist"
```

Add a method to the data model controller to find the Documents directory and return a URL for the path to our file.

```
func getDataFile(datafile: String) -> URL {  
    //get path for data file  
    let dirPath = FileManager.default.urls(for: .documentDirectory, in:  
.userDomainMask)  
    let docDir = dirPath[0] //documents directory  
    print(docDir)  
  
    // URL for our plist  
    return docDir.appendingPathComponent(datafile)  
}
```

Now let's add a method to write the data to our data plist.

```
func writeData(){  
    // URL for our plist  
    let dataFileURL = getDataFile(datafile: datafilename)  
    print(dataFileURL)  
    //creates a property list decoder object  
    let plistencoder = PropertyListEncoder()  
    plistencoder.outputFormat = .xml  
    do {  
        let data = try plistencoder.encode(allData.self)  
        try data.write(to: dataFileURL)  
    } catch {  
        // handle error  
        print(error)  
    }  
}
```

Now when we load the data we need to check to see if our data plist exists. If it does, we'll use that, if not we'll use our default plist.

```

Update loadData()
    func loadData(){
        let pathURL:URL?

        // URL for our plist
        let dataFileURL = getDataFile(datafile: datafilename)
        print(dataFileURL)

        //if the data file exists, use it
        if FileManager.default.fileExists(atPath: dataFileURL.path){
            pathURL = dataFileURL
        }
        else {
            // URL for our plist
            pathURL = Bundle.main.url(forResource: fileName, withExtension:
"plist")
        }

        //creates a property list decoder object
        let plistdecoder = PropertyListDecoder()
        do {
            let data = try Data(contentsOf: pathURL!)
            //decodes the property list
            allData = try plistdecoder.decode([ContinentsData].self,
from: data)
        } catch {
            // handle error
            print(error)
        }
    }

```

Lastly in ContinentTableViewController.swift we need to save our data when the app moves out of the active state.

```

Update viewDidLoad()
    //application instance
    let app = UIApplication.shared
    //subscribe to the UIApplicationWillResignActiveNotification notification
    NotificationCenter.default.addObserver(self, selector:
#selector(ContinentTableViewController.applicationWillResignActive(_:)),
name: UIApplication.willResignActiveNotification, object: app)

```

Add the method called when the notification is received. @objc is needed in Swift 4 to specifically expose the method to Objective-C

```

//called when the UIApplicationWillResignActiveNotification notification is posted
//all notification methods take a single NSNotification instance as their argument
@objc func applicationWillResignActive(_ notification: NSNotification){
    continentsData.writeData()
}

```

Remember that to test this data persistence you need to go to the home screen and then stop the running

app in Xcode (or kill the app on a device) and then run it again.