ATLS 4320: Advanced Mobile Application Development

Week 4: Navigation Controllers

Navigation Controllers

Navigation controllers manage the navigation of hierarchical content by providing a drill-down interface for hierarchical data. https://developer.apple.com/documentation/uikit/uinavigationcontroller. Often used with table views. On the iPhone and iPod hierarchical data is best shown using a succession of table views.

The **UINavigationController** class has two main components

- Stack of controllers
- Navigation bar

The root controller is the initial view controller a navigation controller displays.
Subsequent view controllers are sub-controllers

Navigation Bars

The **UINavigationBar** class enables the navigation bar to manage navigation between different views in a navigation controller https://developer.apple.com/documentation/uikit/uinavigationbar
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 (new iOS11)
- 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 make sure it's used intentionally.
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 UINavigationBar **prefersLargeTitles** property to true (default is false). https://developer.apple.com/documentation/uikit/uinavigationbar/2908999-preferslargetitles

Then you can use the UINavigationItem **largeTitleDisplayMode** property to configure the title's appearance https://developer.apple.com/documentation/uikit/uinavigationitem.largetitledisplaymode so you can control at each level if the large size or the standard size title is used.

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 (countries)

File -> New Project
Single View app
iPhone
countries

Root view controller

Go into the MainStoryboard document outline and **delete** the view.
Drag a Table View out into the controller.
Go into the connections inspector and **connect** the dataSource and delegate to the View Controller icon.
Drag out a table view cell onto the view.
Select the Table View Cell and in the attributes inspector **make** the identifier “CountryCell”.
We want our class to be the controller so go into ViewController.swift and **change** its super class to `UITableViewController`.
Back in storyboard, select the View Controller and **go** into the identity inspector and make sure the class is our ViewController class.
(Or you can do the same thing we did last time, replace the view controller with a table view controller.)

Now for this table view controller to be controlled by a navigation controller, with the controller selected go to **Editor -> Embed in -> Navigation Controller**.
This creates a navigation controller as the root view controller.
Be sure to **check** “is initial view controller” on the navigation controller in the Attributes Inspector.
It also created a *relationship segue* from the navigation controller to the table view controller.

Select the navigation item in the table view controller and **give** it a title (“Continents”) in the Attributes inspector

If you run it you should see an empty table (and a navigation bar at the top). Although this looks the same as last time, we’re going to be able to navigate from the cells to another view controller (once we add that).

Drag `continents2.plist` into your app and *make sure you have Copy items if needed checked*.
It’s an Array of dictionaries that have key continent, value String, key countries, value array of Strings.

Before we get our table set up we’re going to **create** a struct for our data model.
File -> New File
iOS Source Swift File
Continents
Make sure it’s saving to your project folder and the target is checked.

```
//need to conform to Codable protocol since we'll be encoding and decoding
struct ContinentsDataModel: Codable {
    var continent: String
    var countries: [String]
}
```

Now we’ll **add** a class to control our data model (same file). This is similar to our music picker app with the dependent picker, but we also include methods to add and delete countries.

```
class ContinentsDataController {
    var allData = [ContinentsDataModel]()
    let fileName = "continents2"

    //load data from plist
    func loadData() throws {
        //check for file and get URL if possible
        if let dataURL = Bundle.main.url(forResource: fileName, withExtension:
"plist") {
            let decoder = PropertyListDecoder()
            do {
                //get byte buffer (raw data)
                let data = try Data(contentsOf: dataURL)
                //decode to our model
                allData = try decoder.decode([ContinentsDataModel].self, from:
```

```
data)
            } catch {
                throw DataError.CouldNotDecode
            }
        }
        else {
            //couldn't get path
            throw DataError.NoDataFile
        }
    }

    //fetch all the continents
    func getContinents() -> [String] {
        //init empty array
        var allContinents = [String]()
        //loop through data and append to array
        for item in allData {
            allContinents.append(item.continent)
        }
        return allContinents
    }

    //get array of countries based on continent
    func getCountries(idx: Int) -> [String] {
        return allData[idx].countries
    }

    //add a country
    func addCountry(dataIdx: Int, newCountry: String, countryIdx: Int) {
        allData[dataIdx].countries.insert(newCountry, at: countryIdx)
    }

    //delete a country
    func deleteCountry(dataIdx: Int, countryIdx: Int) {
        allData[dataIdx].countries.remove(at: countryIdx)
    }
}
```

In ViewController.swift **create** an instance of the ContinentsDataModelController class to load and access our data and an array for the list of continents.

```
class ViewController: UITableViewController {
    var continentsList = [String]()
    var continentsDataController = ContinentsDataController()
```

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()
    // Do any additional setup after loading the view.
    do {
        try continentsDataController.loadData()
        continentsList = continentsDataController.getContinents()
    } catch {
        print(error)
    }
}
```

Now let's get our table set up. This process will be similar to our last app.
We'll **implement** the two required methods for the UITableViewDataSource protocol.

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    return continentsList.count
}

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "CountryCell", for:
indexPath)
    cell.textLabel?.text = continentsList[indexPath.row]
    return cell
}
```

Run it and you should see the continent data listed in the table view.

To enable large titles, **add** to `viewDidLoad()`

```
//enables large tiles
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
Name it *DetailViewController* and **subclass** of UITableViewController
Once created double check that DetailViewController 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** that the class for your new table view controller.
Before we forget, select the table view cell in the detail view controller and **give** it the identifier CountryCell.

Now let's make the segue from the Root to the Detail View Controller.
Ctrl-click from the master prototype cell and **drag** to the detail view controller.
When you release the mouse, 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 *Master View Controller* **select** the table view cell and in the attributes inspector **change** accessory to disclosure indicator. This indicates that selecting that row will bring up related data.
We're not going to set the title for this view in the storyboard since 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 set Large Title to Never.

If you run it at this point the controller will navigate, we just have to load the data.

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

```
class DetailViewController: UITableViewController {

    var continentsData = ContinentsDataController()
    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(idx: selectedContinent)
}
```

Now let's **update** the delegate protocol methods for our TableView. 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, numberOfRowsInSection section:
```

```
Int) -> Int {
    // #warning Incomplete implementation, return the number of rows
    return countryList.count
}


override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "CountryCell", for:
indexPath)

    // Configure the cell...
    cell.textLabel?.text = countryList[indexPath.row]

    return cell
}
```

The last part is for the Master View Controller to set data in the Detail View Controller.
Go into ViewController.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 ContinentsDataModelController instance. The place to do this is `prepareForSegue`, as it's about to transition to the detail view controller.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "CountrySegue" {
        //get reference to DetailViewController (need to downcast from
UIViewController)
        let detailVC = segue.destination as! DetailViewController
        //get the cell that triggered the segue (need to downcast)
        let indexPath = tableView.indexPath(for: sender as! UITableViewCell)
        //set data in destination controller
        if let selection = indexPath?.row {
            detailVC.selectedContinent = selection
            detailVC.title = continentsList[selection]
            detailVC.continentsData = continentsDataController
        }
    }
}
```

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.

Delete Rows

Now let's add the ability to delete countries.

In DetailViewController.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 to support conditional editing of the table view.
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 to support editing the table view.
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        //Notify data model
        countryList.remove(at: indexPath.row)
        //Update instance
        continentsData.deleteCountry(dataIdx: selectedContinent, countryIdx:
indexPath.row)
        //Update table
        tableView.deleteRows(at: [indexPath], with: .fade)
    } 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
    }
}
```

Move Rows

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

```
// Override to support conditional rearranging of the table view.
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 to support rearranging the table view.
override func tableView(_ tableView: UITableView, moveRowAt fromIndexPath:
IndexPath, to: IndexPath) {
    //get idx of rows
    let fromRow = fromIndexPath.row
    let toRow = to.row

    //get name of country being moved
    let moveCountry = countryList[fromRow]

    //swap in array
    countryList.swapAt(fromRow, toRow)

    //swap in data controller
    continentsData.deleteCountry(dataIdx: selectedContinent, countryIdx: fromRow)
    continentsData.addCountry(dataIdx: selectedContinent, newCountry:
moveCountry, countryIdx: 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 as well.
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.
Do the following for auto layout:

1. Embed label and textfield into stackview
2. Set content hugging and content resistance to 249 and 749 respectively for the textfield in the size inspector
3. Pin the stackview to leading and trailing (8 constant) and top (whatever you want)

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 DetailViewController.swift and in ViewDidLoad comment out

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

Back in the storyboard go into the Detail view and **add** a navigation item. Remove its title.
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 Detail view controller to the AddCountry view controller and **choose** a *Present Modally* segue.
We don't want a 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 AddCountry view controller, 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 AddCountry view controller.
Make its title Add New Country
Add two bar button items.
Put one on the right and **change** it's system item to Save, and the other on the left and **change** it's system item to Cancel.
You can also **change** the bar tint to white to match the nav bar in the detail controller.

If you run it at this point you should see the nav bar with the buttons but they don't do anything yet.
Add constraints to navigation bar – leading, trailing, and top.

Let's **implement** the Cancel button by setting up an unwind segue to undo the modal segue back to the detail view controller.
First we have to create an unwind method in the destination view controller, DetailViewController. 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 on top of the view controller and choose this unwind method.
Choose this segue and **give** it the identifier "cancel".
Do the same thing for the Done button and **give** it's segue the identifier "save".
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.

```
class AddCountryViewController: UIViewController {

    //connect textfield
    @IBOutlet weak var countryTextField: UITextField!

    //var to store user input
    var addedCountry = String()
```

When the user taps Done we want to get the value from the textfield.
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 **add** the .isEmpty in the prepare for segue method as a check to avoid adding empty rows.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    //only care if they hit save
    if segue.identifier == "save" {
        //make sure they entered info
        if countryTextField.text?.isEmpty == false {
            addedCountry = countryTextField.text!
        }
    }
}
```

In DetailViewController we **implement** the unwind method

```
//called automatically
@IBAction func unwindSegue(_ segue: UIStoryboardSegue) {
    //check the id of segue
    if segue.identifier == "save" {
        //downcast to access members
        let source = segue.source as! AddCountryViewController

        //double check to make sure new country name is not empty
        if source.addedCountry.isEmpty == false {
            //add new country to data model (notify of changes)
            continentsData.addCountry(dataIdx: selectedContinent, newCountry:
source.addedCountry, countryIdx: countryList.count)
            //add to working copy
            countryList.append(source.addedCountry)
            //update table view based on data changes
            tableView.reloadData()
        }
    }
}
```

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(master) cell to the new country info view controller and choose Accessory Action Show segue. Give it the identifier "ContinentSegue.

In the continents scene(master) **change** the accessory to a detail disclosure in the table view cell. This is because selecting a row will still drill down to the countries, but tapping on the accessory will bring us to our new table view.
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".

Now we're going to **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 and **add** two variables

```
class ContinentInfoTableViewController: UITableViewController {

    //properties to hold info
    var continent = String()
    var number = String()
```

Delete the 2 dataSource methods because static cells don't use them.
Now if you run it you can see your static cells.

Now let's **populate** them with data.

```
override func viewWillAppear(_ animated: Bool) {
```

```
    continentName.text = continent
    countryNumber.text = number
}
```

In ViewController.swift we have to update `prepareForSegue` to work with the detail disclosure accessory button by adding the following after the if statement:

```
else if segue.identifier == "ContinentSegue" {
            //get access to destination controller (need to downcast)
            let infoVC = segue.destination as! ContinentInfoTableViewController
            //get the selected cell
            let indexPath = tableView.indexPath(for: sender as! UITableViewCell)
            //set the continent name in destination
            infoVC.continent = continentsList[indexPath!.row]
            //get the country list
            let countryList = continentsDataController.getCountries(idx:
indexPath!.row)
            //set number of countries (cast integer to string)
            infoVC.number = String(countryList.count)
            infoVC.title = continentsList[indexPath!.row]
}
```

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 by adding to `viewDidLoad()`
`navigationItem.largeTitleDisplayMode = .never`

Data Persistence (countries data)

Let's **add** data persistence to our countries app.
In the ContinentsDataModelController 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

    // URL for our plist
    return docDir.appendingPathComponent(datafile)
}
```

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

```
func writeData() throws {
    let dataFileURL = getDataFile(datafile: dataFileName)
    //get an encoder
    let encoder = PropertyListEncoder()
    //set format -- plist is a type of xml
    encoder.outputFormat = .xml
    do {
        let data = try encoder.encode(allData.self)
        try data.write(to: dataFileURL)
    } catch {
        print(error)
        throw DataError.CouldNotEncode
    }
}
```

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() throws {
    let pathURL: URL?

    //get the path where our data file would be
    let dataFileURL = getDataFile(datafile: dataFileName)

    //check to see if the data file exists
    if FileManager.default.fileExists(atPath: dataFileURL.path) {
        pathURL = dataFileURL
    } else {
        //load default data if we can't find a user data file
        pathURL = Bundle.main.url(forResource: fileName, withExtension: "plist")

    }

    //check for file and get URL if possible
    if let dataURL = pathURL {
        let decoder = PropertyListDecoder()
        do {
            //get byte buffer (raw data)
            let data = try Data(contentsOf: dataURL)
            //decode to our model
            allData = try decoder.decode([ContinentsDataModel].self, from: data)
        } catch {
            throw DataError.CouldNotDecode
```

```
        }
    }
    else {
        //couldn't get path
        throw DataError.NoDataFile
    }
}
```

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

Update viewDidLoad()

```
//get app instance
let app = UIApplication.shared

//subscribe to willResignActive notification
NotificationCenter.default.addObserver(self, selector:
#selector(ViewController.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 automatically when UIApplicationWillResignActive notification is posted
for our app
//needs to take an NSNotification as parameter
@objc func applicationWillResignActive(_ notification: NSNotification) {
    do {
        try continentsDataController.writeData()
    } catch {
        print(error)
    }
}
```

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.