ATLS 4320: Advanced Mobile Application Development

Week 5: Collection Views

Collection Views

Table views were being used to display such a huge variety of interfaces that in iOS 6 Apple introduced collection views which are more customizable than table views.
https://developer.apple.com/design/human-interface-guidelines/ios/views/collections/

- Collection views makes it easy to display data in rows and columns to form grids
- Similar approach as the table view data source and delegate pattern
- Collection views are very customizable
- You can create your own custom layout and transitions
- Can scroll horizontally or vertically

The collection view is the main view that manages and displays an ordered collection of items

- **UICollectionView** class
- **UICollectionViewDataSource** protocol handles the data for a collection view
- **UICollectionViewDelegate** protocol manages the selection of items in a collection view

Cells hold the content in the view

- **UICollectionViewCell** class

Supplementary views hold extra information that you want but not in the cells

- Header
  - **UICollectionElementKindSectionHeader**
- Footer
  - **UICollectionElementKindSectionFooter**
- **UICollectionReusableView** class is used for cells and supplementary views in a collection view

Sections, items, and cells are handled similar to table views and have similar methods to handle them.

Layout

In order to allow for more flexible layout, the UICollectionView class doesn't handle the layout of collection views, it uses a helper class to do that

- The **UICollectionViewLayout** class is an abstract base class that generates layout information for a collection view. You **must** subclass it in order to use it. The layout object provides layout information such as position and size of the elements to the views being created by the collection view's data source.
- Collection views have **three types of visual elements** that need to be laid out
  - **Cells**: a single data item in the collection
  - **Supplementary views:** header and footer views. These cannot be selected by the user and are optional.
  - **Decoration views**: visual adornments not related to the data. These cannot be selected by the user and are optional.

- Instead of subclassing UICollectionViewLayout, you might be able to use the **UICollectionViewFlowLayout** class, a subclass of UICollectionViewLayout that you can use to create a flow layout for a collection view
  - scrollDirection property determines direction the view scrolls; default is vertical
  - configure header and footer (supplementary) views
- The **UICollectionViewDelegateFlowLayout** protocol methods let you dynamically define the size and spacing of items, sections, headers and footers. If you don't implement the methods in this delegate, the flow layout will use the default values in the UICollectionViewFlowLayout class.
- Create your own subclass of UICollectionViewLayout if UICollectionViewFlowLayout doesn't meet your needs

UIEdgeInsets is a struct in the UIKit framework that allows you to set edge inset values to change the area represented by a rectangle. (slide) https://developer.apple.com/documentation/uikit/uiedgeinsets

pictureCollection Demo App

(pictureCollection)
File | New Project
Single View App

Delete *ViewController.swift*
Delete *the view controller* in the storyboard
Create a new *Cocoa Touch class* called **CollectionViewController** and **subclass** *UICollectionViewController*.
We could have also changed the superclass in the file that the template created but this gives us all the stub methods we'll need.
The delegate methods included are very similar to the table view delegate methods we've been using.

Add a new *Collection View Controller* in the storyboard and **check** *Is Initial View Controller*.
Change its class to be our new *CollectionViewController* class.
Note that the UICollectionViewController has also brought with it a collection view and a prototype collection view cell.
Select the collection view and in the connections inspector *make sure* the delegate and data source are connected to the Collection View Controller.

Create a new Cocoa Touch class called CollectionViewCell and **subclass** UICollectionViewCell.
In the storyboard **click** on the collection view cell and **change** its class to CollectionViewCell.
Also **give it a reuse identifier "Cell"** (must match the constant in CollectionViewController).
Since our cells are going to hold images, **make** the prototype cell a bit larger and **drag** an image view into it.
For the cell size, **choose** custom and make the size 100x100 and **make** the image view 100x100.
With the Image View **selected** in the storyboard scene **Pin** the Spacing to nearest neighbor constraints on all four sides of the view to 0 with the Constrain to margins option unchecked. **Create** 4 constraints.

Create a connection from the image view called imageView but make sure you're connecting the imageview to the CollectionViewCell class, **NOT** the CollectionViewController class. If you get an error and it's the wrong class delete the connection and the variable and redo it to the right class.

Images

Add the images to the project by dragging them into the Assets folder.
Note that they're named album1 through album20. My code will count on this naming scheme.

In CollectionViewController.swift **define** an array to hold the image names.

```
class CollectionViewController: UICollectionViewController,
UICollectionViewDelegateFlowLayout {

    //array for names of all the album images
    var albumImages = [String]()
```

In `viewDidLoad()` **comment out** this line as we set the reuse identifier in the storyboard (either one is fine but you don't need both)

```
self.collectionView!.registerClass(UICollectionViewCell.self,
forCellWithReuseIdentifier: reuseIdentifier)
```

Update `viewDidLoad()` to load the image names into the array. I named my images the same with sequential numbers so I could use a for loop.

```
override func viewDidLoad() {
    super.viewDidLoad()

    //populate array with all the image names
    for i in 1...20 {
        albumImages.append("album" + String(i))
    }
}
```

Data source methods

Update the data source methods which you'll notice are very similar to the table view data source methods. For now we have one section.

```
// MARK: UICollectionViewDataSource
override func numberOfSections(in collectionView: UICollectionView) -> Int {
    return 1
}

override func collectionView(_ collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {
    return albumImages.count
}
```

```
override func collectionView(_ collectionView: UICollectionView, cellForItemAt
indexPath: IndexPath) -> UICollectionViewCell {
    //get the cell and downcast to access the imageView property from our custom
cell class
    let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
reuseIdentifier, for: indexPath) as! CollectionViewCell

    //construct UIImage and set imageView source
    let image = UIImage(named: albumImages[indexPath.row])
    cell.imageView.image = image

    return cell
}
```

Run the app and the images should appear.

Cell Item Size and Spacing

All the images are a different size so we need to set the size of each cell to match the image.
We can use a method in the UICollectionViewDelegateFlowLayout protocol to get the size of each image.
Adopt the UICollectionViewDelegateFlowLayout protocol

```
class CollectionViewController: UICollectionViewController,
UICollectionViewDelegateFlowLayout {
```

Set some constants to use for laying out the collection view's cells

```
let spacing : CGFloat = 8
let numberOfItemsPerRow : CGFloat = 3
let spacingBetweenCells : CGFloat = 8
```

Add the following to viewDidLoad() to use a flow layout and set some of the instance properties.

```
//create a layout instance and set properties
let layout = UICollectionViewFlowLayout()
layout.sectionInset = UIEdgeInsets(top: spacing, left: spacing, bottom: spacing,
right: spacing)
layout.minimumLineSpacing = spacing
layout.minimumInteritemSpacing = spacing
collectionView.collectionViewLayout = layout
```

Implement the method that gets the size of the item's cell. When this method is not implemented it just uses the size in the item's size property.

```
//size equally and space equally
func collectionView(_ collectionView: UICollectionView, layout
collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath)
```

```
-> CGSize {
        //Amount of total amount of white space in a row including margin
        let totalSpacing = (2 * spacing) + ((numberOfItemsPerRow - 1) *
spacingBetweenCells)

        //get size of each item
        let width = (collectionView.bounds.width -
totalSpacing)/numberOfItemsPerRow
        return CGSize(width: width, height: width)
    }
```

Run the app and see how it's all evenly spaced. Feel free to change the constants to see more items on each row or increase the spacing.

Header

In the storyboard **click** on the Collection View and in the attributes inspector next to Accessories **check** Section Header.
This adds a section header that you can configure.
Add a label and **add** constraints as needed.

Now we **need** a class to control the new header.
File -> New File -> Cocoa Touch Class
Name it *HeaderSupplementaryView* and **subclass** *UICollectionReusableView*.

Then go back into the storyboard, **select** the collection reusable view and **change** its class to the class you just created *HeaderSupplementaryView*
Also in the attributes inspector **give** the header a reuse identifier "*Header*"
Create an outlet for the label called *headerLabel*. **Make sure you're connecting it to the HeaderSupplementaryView class**.

Go into CollectionViewController.swift and **implement** the following data source methods to set the size and content of the header.

```
//set height for header
func collectionView(_ collectionView: UICollectionView, layout
collectionViewLayout: UICollectionViewLayout, referenceSizeForHeaderInSection
section: Int) -> CGSize {
    return CGSize.init(width: 50, height: 50)
}

//set content for header
override func collectionView(_ collectionView: UICollectionView,
viewForSupplementaryElementOfKind kind: String, at indexPath: IndexPath) ->
UICollectionReusableView {

    var header: HeaderSupplementaryView?
    if kind == UICollectionView.elementKindSectionHeader {
        header = collectionView.dequeueReusableSupplementaryView(ofKind: kind, 
withReuseIdentifier: "Header", for: indexPath) as? HeaderSupplementaryView
        header?.headerLabel.text = "2019"
    }

    return header!
}
```

Now when you run it you should see your header.
You can do the same thing for a footer and use the same method but add a check for UICollectionElementKindSectionFooter to configure it.

Detail

Now let's add a detail view to show the image larger when the user taps on a cell.

Embed the collection view controller in a navigation controller.
You'll notice that along with a navigation controller it adds a navigation item.
In the navigation item make the title "Top Albums".

If you want large titles **add** to viewDidLoad()

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

Drag a view controller into the storyboard.
Create a *show segue* from the collection view cell to the new view controller. To ensure the connection is from the cell, use the document outline to create the segue.
Give the segue an identifier called *showDetail*.
Add an image view to take up the full view, all the way up to the navigation bar.
Change its mode to Aspect Fit (or fill)
Add constraints so the image view fills up the view.

Add a new Cocoa touch class called DetailViewController and subclass UIViewController.
Back in the storyboard **make** this the class for the new view controller.
Go into the assistant editor and **connect the image view as an outlet called imageView**. Make sure you **make** the connection to `DetailViewController.swift`

In `DetailViewController.swift` add a string to hold the name of the image passed to this view.

```
var imageName : String?
```

Add the following to populate the image view with the image the user selected every time the detail view appears.

```
//load the image every time the view appears
override func viewWillAppear(_ animated: Bool) {
    if let name = imageName {
        imageView.image = UIImage(named: name)
    }
}
```

imageName is an optional and we conditionally unwrap so we handle the case of no image being passed. If we assume it always has a value and it doesn't, the app would crash.

In `CollectionViewController.swift` we need to figure out which cell the user selected and then pass that data to `DetailViewController` when the segue occurs.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // Get the new view controller using [segue destinationViewController].
    // Pass the selected object to the new view controller.
    if segue.identifier == "showDetail" {
        //reference to destination
        let detailVC = segue.destination as! DetailViewController
        //get the cell that is triggering the segue
        let indexPath = collectionView.indexPath(for: sender as!
CollectionViewCell)
        //set title of view controller
        detailVC.title = "Album #\(indexPath!.row)"
        //set the image name property
        detailVC.imageName = albumImages[indexPath!.row]
    }
}
```

Sharing

Now we want to be able to share an image from the detail view controller.

In the storyboard on the Detail View Controller scene **add** a bar button item onto the right side of the navigation bar and **set** the system item property to *action* to get the sharing icon.
Create an action for the button called share(). Make sure you are connecting the button to the DetailViewController class.

Implement this function in `DetailViewController.swift`

```
//implement sharing functionality
@IBAction func share(_ sender: Any) {
    var imageArray = [UIImage]()
    imageArray.append(imageView.image!)

    //create the instance of activity view controller
    let shareScreen = UIActivityViewController(activityItems: imageArray,
applicationActivities: nil)
    shareScreen.modalPresentationStyle = .popover
    present(shareScreen, animated: true, completion: nil)
}
```

We create an array even though we're just sharing one image because the activityItems parameter in the `UIActivityViewController` initializer expects an array.

The `UIActivityViewController` class lets you provide several standard services from your app, including the ones you see in a share menu. We are responsible for configuring, presenting, and dismissing this view controller.
The modalPresentationStyle property is a property on the UIViewController class
The popover presentation controller manages the display of content in a popover. We assign the activity view controller to the popover bar button item to anchor the popover.

Security

To protect user privacy, you must declare ahead of time any access to private data or your App will crash.
Frameworks that count as private data:
Contacts, Calendar, Reminders, Photos, Bluetooth Sharing, Microphone, Camera, Location, Health, HomeKit, Media Library, Motion, CallKit, Speech Recognition, SiriKit, TV Provider.

To access the user's photo library you must include the `NSPhotoLibraryAddUsageDescription` key in your app's `Info.plist` file and provide a purpose string for this key. If your app attempts to access the user's photo library without a corresponding purpose string, your app exits.

In Info.plist **click** on the + and start typing "Privacy - Photo Library Additions Usage Description"
Assign a string that explains to the user why this access is required.