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 content in a visual layout
- Well suited for items that vary in size such as images
- Collection views are very customizable
- You can create your own custom layout and transitions
- Can scroll horizontally or vertically
- Similar approach as the table view data source and delegate pattern

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

- **UICollectionView** class https://developer.apple.com/documentation/uikit/uicollectionview
- **UICollectionViewDataSource** protocol handles the data for a collection view https://developer.apple.com/documentation/uikit/uicollectionviewdatasource
    - Adopt the UICollectionViewDataSource protocol and implement the required methods
    - Use a **UICollectionViewDiffableDataSource** (added in iOS13) object as your data source object, which already conforms to this protocol or create a custom data source object. https://developer.apple.com/documentation/uikit/uicollectionviewdatasource
        - manages updates to your collection view's data source and user interface
        - abstracts a significant amount of the logic in UICollectionViewDataSource and results in less code
        - automatic data synchronization
        - automatic data change animations
- **UICollectionViewDelegate** protocol manages the selection of items in a collection view https://developer.apple.com/documentation/uikit/uicollectionviewdelegate

Data in the collection view is organized into individual items which are cells

- **UICollectionViewCell** class https://developer.apple.com/documentation/uikit/uicollectionviewcell

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

- Header **.elementKindSectionHeader** property
- Footer **.elementKindSectionFooter** property
- **UICollectionReusableView** class is used for cells and supplementary views in a collection view https://developer.apple.com/documentation/uikit/UICollectionReusableView

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. A layout object is used to define the visual arrangement of the content in the collection view.

- The layout is stored in the collection view's collectionViewLayout property
- 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.

The **UICollectionViewLayout** class is an abstract base class that you subclass to define the organization and location of all cells and supplementary views inside the collection view.

- 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. https://developer.apple.com/documentation/uikit/uicollectionviewlayout
- Instead of subclassing UICollectionViewLayout, you might be able to use the **UICollectionViewFlowLayout** class, a subclass of UICollectionViewLayout provided that you can use to create a flow layout for a collection view https://developer.apple.com/documentation/uikit/uicollectionviewflowlayout
    - 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. https://developer.apple.com/documentation/uikit/uicollectionviewdelegateflowlayout
- The **UICollectionViewCompositionalLayout** (added in iOS13) class makes it simpler to build complex layouts https://developer.apple.com/documentation/uikit/uicollectionviewcompositionallayout
    - Composable – lets you build a layout made up of the components by building up from items into a group, from groups into a section, and finally into a full layout
        - An item is usually a single cell but can also be headers, footers, etc
            - **NSCollectionLayoutItem** lets you define an item's size, space, and layout
        - Groups provide a container for a set of items and determine how the items in a collection view are laid out in relation to each other.
            - **NSCollectionLayoutGroup** let you define the group's size, orientation, the items in the group, and the number of items in the group
        - Sections provide a container to combine a set of groups and provide a way to separate the layout of a collection view into distinct pieces. Each section can have the same, or different, layouts from each other.
            - **NSCollectionLayoutSection** lets you define the background, header, and footer of the section
    - These all use **NSCollectionLayoutSize** for the width and height
        - **NSCollectionLayoutDimension** defines the dimension
        - Value can be expressed as absolute, estimated, or fractional
- Create your own subclass of **UICollectionViewLayout**

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

Expo
File | New Project
iOS App
Product Name: Expo

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

Delete ViewController.swift (and move to trash)
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 in the attributes inspector 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, a collection view cell, and a collection view flow layout.
Select the collection view and in the connections inspector make sure the delegate and data source are connected to the Collection View Controller. In the attributes inspector note that the Layout is set to Flow.

Create a new Cocoa Touch class called CollectionViewCell and subclass UICollectionViewCell.
In the storyboard click on the collection view cell and in the identity inspector change its class to CollectionViewCell.
In the attributes inspector give it a reuse identifier "Cell" (must match the identifier used in CollectionViewController).
Since our cells are going to hold images drag an image view into the cell.
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 an outlet connection from the image view called imageView but make sure you're connecting the imageView to the CollectionViewCell class, NOT the view controller 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. Instead of opening the Assistant under Editor Options you might just need to add an editor on the right and navigate to the CollectionViewCell class.

<u>Images</u>
Add the images to the project by dragging them into the Assets folder.
Note that they're named atlas1 through atlas20. My code will count on this naming scheme.
In CollectionViewController.swift define an array to hold the image names.
```
var expoImages=[String]()
```

Make sure you have a constant defined for reuseIdentifier.
```
let reuseIdentifier = "Cell"
```

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.

```
        for i in 1...20{
            expoImages.append("atlas" + String(i))
        }
```

<u>Data source methods</u>

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

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

    override func collectionView(_ collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {
        // #warning Incomplete implementation, return the number of items
        return expoImages.count
    }
```

Make sure you cast the cell to the CollectionViewCell class we created.

```
    override func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
reuseIdentifier, for: indexPath) as! CollectionViewCell

        // Configure the cell
        let image = UIImage(named: expoImages[indexPath.row])
        cell.imageView.image = image
        return cell
    }
```

Run the app and the images should appear. Now let's deal with the layout and size of the images.

<u>Layout</u>

We're going to create a function to set up our layout and return a UICollectionViewLayout object that we'll use for our collection view layout.

```
    func generateLayout() -> UICollectionViewLayout {
        // create item size
        let itemSize = NSCollectionLayoutSize(widthDimension:
.fractionalWidth(1.0), heightDimension: .fractionalHeight(1.0))
        // create an item layout
        let photoItem = NSCollectionLayoutItem(layoutSize: itemSize)
        // create group size
        let groupSize = NSCollectionLayoutSize(widthDimension:
.fractionalWidth(1.0), heightDimension: .fractionalHeight(0.2))
        // create a group arranged horizontally
        let group = NSCollectionLayoutGroup.horizontal(layoutSize:
groupSize, subitem: photoItem, count: 1)
        // create a section with one group
        let section = NSCollectionLayoutSection(group: group)
        // create and return the layout object
        let layout = UICollectionViewCompositionalLayout(section: section)
        return layout
    }
```

The fractional sizes are relative to their container. So an item with a fractional width and height of 1 means it will fill the group that contains it. Using fractional width and height means that the image aspect ratio will be preserved.
When creating the group the count parameter determines how many items are in the group.
In this layout the section only has 1 group.

In viewDidLoad() we will call this method and use the UICollectionViewCompositionalLayout object as the layout for our collection view.

```
collectionView.collectionViewLayout = generateLayout()
```

Now when you run it the images are sized better. We can easily change the layout by changing how many items are in a group. Change it to 2 or 3 and see how the layout changes.

Spacing
We can use the **NSCollectionLayoutItem** contentInsets property to control the space around each item.
In generateLayout() after creating the layout item we can set the contentInsets property.

```
photoItem.contentInsets = NSDirectionalEdgeInsets(top: 10, leading: 10,
bottom: 0, trailing: 10)
```

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.
In the attributes inspector give the header a reuse identifier “header”
Add a label and add constraints as needed.
If you made the background black you need to make the text of the label white.
Now we need a class to control the new header.
File | New File | Cocoa Touch class called CollectionSupplementaryView and subclass UICollectionReusableView.
Then go back into the storyboard, select the collection reusable view and in the identity inspector change its class to the CollectionSupplementaryView class you just created.
Create an outlet for the label called headerLabel. Make sure you’re connecting it to the CollectionSupplementaryView class.

Go into CollectionViewController.swift and implement the following data source method.

```
    override func collectionView(_ collectionView: UICollectionView,
viewForSupplementaryElementOfKind kind: String, at indexPath: IndexPath) ->
UICollectionReusableView {
        var header = CollectionSupplementaryView()
        if kind == UICollectionView.elementKindSectionHeader{
            header = collectionView.dequeueReusableSupplementaryView(ofKind:
kind, withReuseIdentifier: "header", for: indexPath) as!
CollectionSupplementaryView
            header.headerLabel.text = "Student Projects"
        }
        return header
    }
```

You can do the same thing for a footer and use the same method but add a check for UICollectionElementKindSectionFooter to configure it.

Why do you think we need to define header where we do and not just in the if statement where we initialize it?

Then we need to add the section header to our layout. Update generateLayout() after you create the section.

```
        // create a header
        let headerSize = NSCollectionLayoutSize(widthDimension:
.fractionalWidth(1.0), heightDimension: .estimated(44))
        let sectionHeader =
NSCollectionLayoutBoundarySupplementaryItem(layoutSize: headerSize,
elementKind: UICollectionView.elementKindSectionHeader, alignment: .top)
        // add the header to the section
        section.boundarySupplementaryItems = [sectionHeader]
```

Now when you run it you should see your header.

<u>Detail</u>
Now let's add a detail view to show the image larger when the user taps on a cell.
First, 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 ATLAS Expo.
If you want large titles add to viewDidLoad()

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

If the collection view has a black background and you've enabled large titles, this will cause the navigation bar to have a black background as well. Select the navigation controller's navigation bar and change the title color under Large Title Text Attributes to white. Without large titles the bar will have the default background color and the text will use the title color in the Title Text Attributes section.

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.
Ensure its mode is Aspect Fit (or fill)
Add constraints so the image view fills up the view. (0 for all pins)
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.

```
    override func viewWillAppear(_ animated: Bool) {
        if let name = imageName {
            imageView.image = UIImage(named: name)
        }
    }
```

imageName is an optional and we conditionally unwrap it 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?) {
        if segue.identifier == "showDetail"{
            let indexPath = collectionView?.indexPath(for: sender as!
CollectionViewCell)
            let detailVC = segue.destination as! DetailViewController
            detailVC.imageName = expoImages[(indexPath?.row)!]
        }
    }
```

Sharing

Now let's add the ability 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 item and set the System Item 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

```
    @IBAction func share(_ sender: UIBarButtonItem) {
        var imageArray = [UIImage]()
        imageArray.append(imageView.image!)
        let shareScreen = UIActivityViewController(activityItems:
imageArray, applicationActivities: nil)
        shareScreen.modalPresentationStyle = .popover
        shareScreen.popoverPresentationController?.barButtonItem = sender
        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 provides 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.
https://developer.apple.com/documentation/uikit/uiactivityviewcontroller

The modalPresentationStyle property is a property on the UIViewController class
https://developer.apple.com/documentation/uikit/uiviewcontroller/1621355-modalpresentationstyle

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 accesses 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 will crash.

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

Now you'll be able to share and save the image to your camera roll.