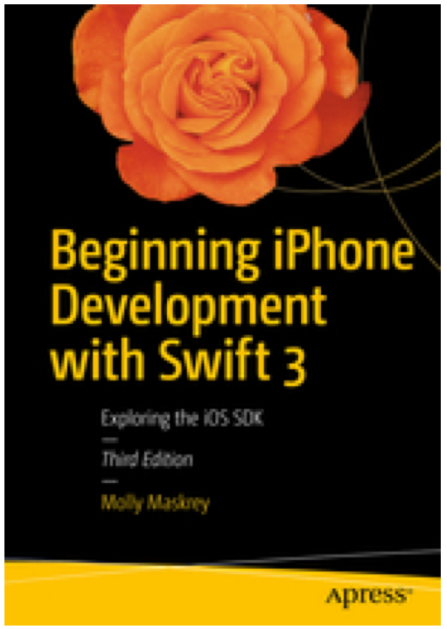
CENTENNIAL COLLEGE



Advanced iOS Development

Week 8

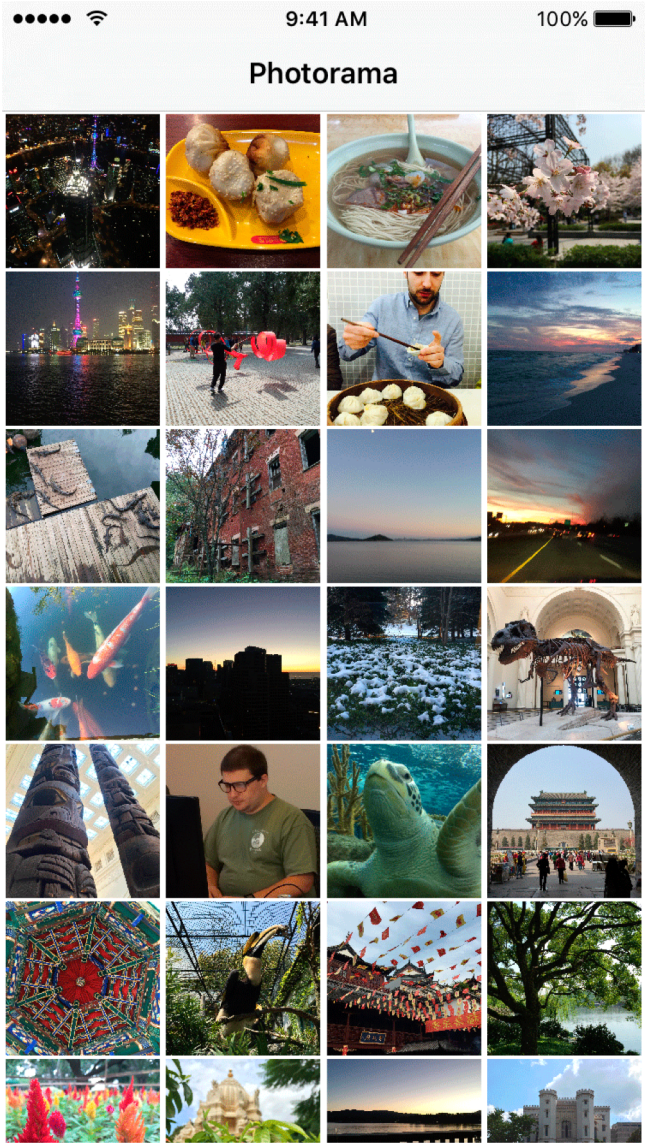
Collection Views

Collection Views Revisited

CENTENNIAL COLLEGE

- In this Lesson, you will continue working on the **Photorama** application by displaying the interesting **Flickr** photos in a grid using the UICollectionView class.

- This lesson will also reinforce the **data source** design pattern.



Collection Views Revisited

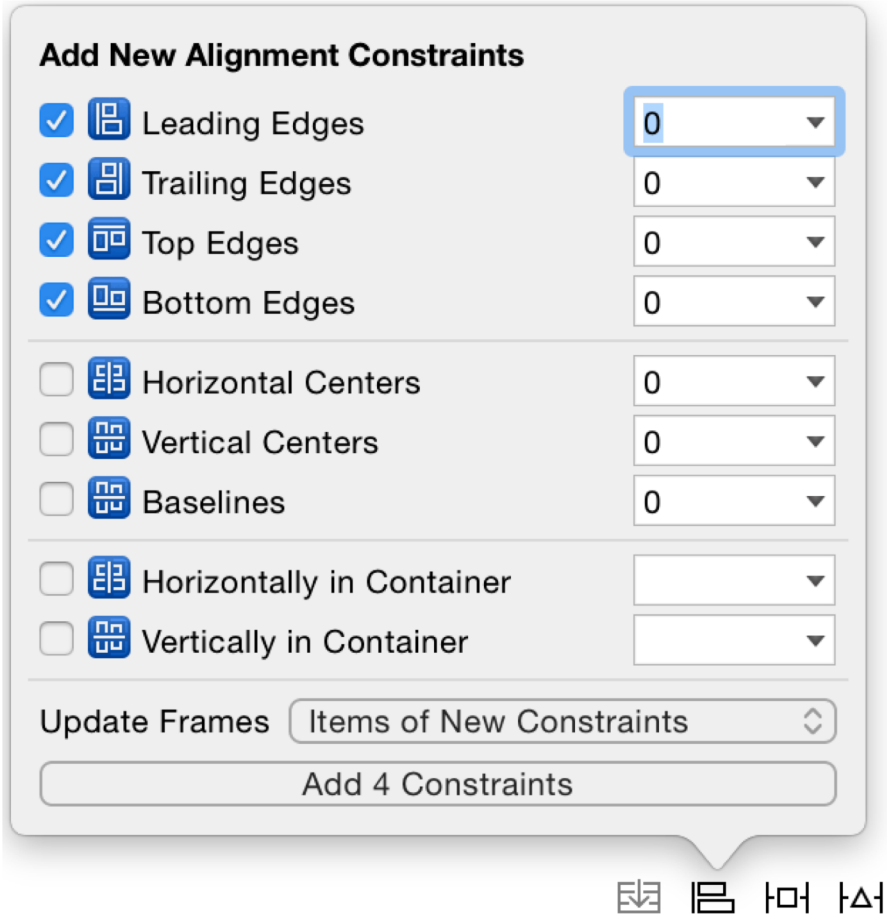
- In a previous lesson, you worked with **UITableView**.
- **Table views** are a great way to display and edit a column of information in a hierarchical list.
- Like a table view, a **collection view** also displays an ordered collection of items, but instead of displaying the information in a hierarchical list, the collection view has a **layout object** that drives the display of information.
- You will use a built-in layout object, the **`UICollectionViewFlowLayout`**, to present the interesting photos in a scrollable grid.

Displaying the Grid

- Let's tackle the interface first.
- You are going to change the UI for **PhotosViewController** to display a collection view instead of displaying the image view.
- Open **Main.storyboard** and locate the **Photorama** image view.
- Delete the image view from the canvas and drag a **Collection View** onto the canvas.
- Select both the **collection view** and its **superview**. (The easiest way to do this is using the document outline.)

Displaying the Grid (continued)

- Open the **Auto Layout Align menu**, configure it like the following Figure, and click Add 4 Constraints .

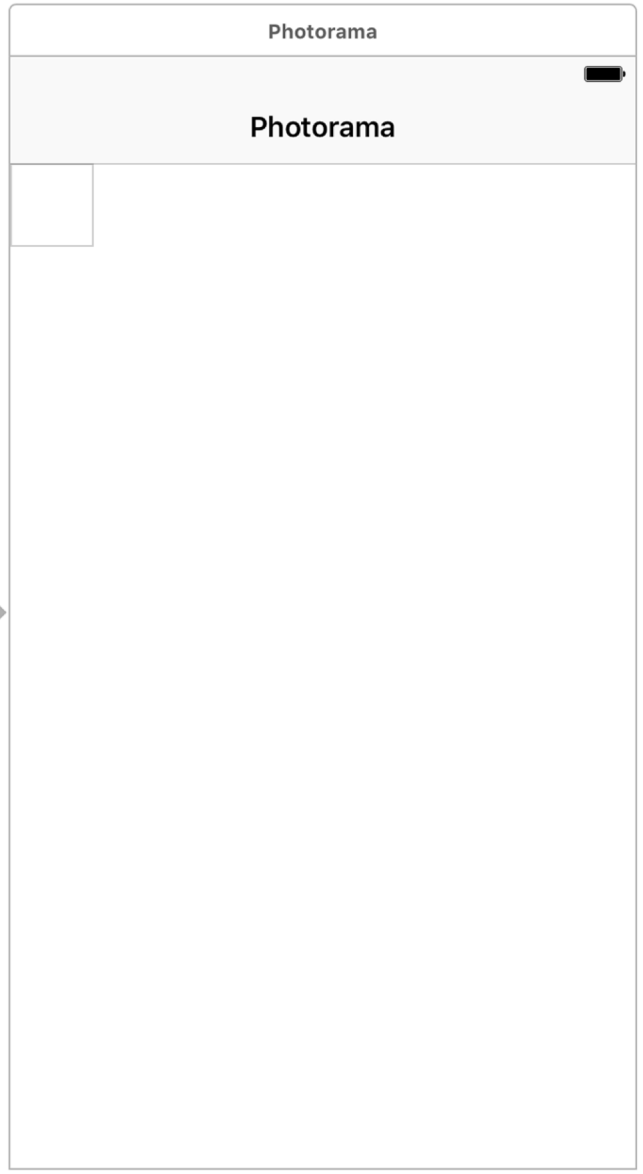


Displaying the Grid (continued)

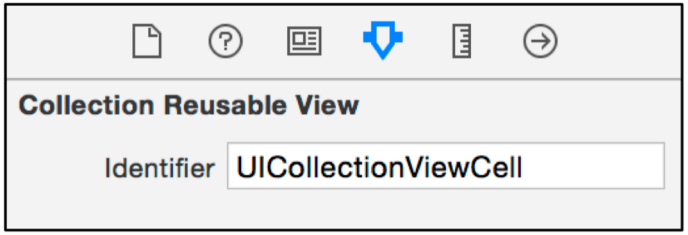
- Because you used the **Align** menu to pin the edges, the collection view will be pinned to the top of the **entire view** instead of to the top layout guide. This is useful for scroll views (and their subclasses, like **`UITableView`** and **`UICollectionView`**) so that the content will scroll underneath the navigation bar.

- The scroll view will automatically update its insets to make the content visible.

- The canvas will now look like Figure 21.3.



Displaying the Grid (continued)



- Currently, the collection view cells have a clear **background color**.
- Select the **collection view cell** – the small rectangle in the upper-left corner of the collection view – and give it a **black background color**.
- Select the black collection view cell and open its **attributes inspector**.
- Set the Identifier to **UICollectionViewCell**.
- The collection view is now on the **canvas**, but you need a way to populate the cells with data.
- To do this, you will create a **new class** to act as the **data source** of the collection view.

Collection View Data Source

- Applications are constantly changing, so part of being a good iOS developer is building applications in a way that allows them to adapt to changing requirements.
- The Photorama application will display a single collection view of photos. You could make the **PhotosViewController** be the **data source** of the collection view.
- The view controller would implement the required data source methods, and everything would work just fine.

Collection View Data Source (continued)

- At least, it would work for now. What if, sometime in the future, you decided to have a different screen that also displayed a collection view of photos?
- Maybe instead of displaying the interesting photos, it would use a different web service to display all the photos matching a search term.
- In this case, you would need to re-implement the same data source methods within the new view controller with essentially the same code. That would not be ideal.
- Instead, you will abstract out the collection view data source code into a new class. This class will be responsible for responding to data source questions – and it will be reusable as necessary.

Collection View Data Source (continued)

- Create a new Swift file named **`PhotoDataSource`** and declare the **`PhotoDataSource`** class.

```
import UIKit

class PhotoDataSource: NSObject, UICollectionViewDataSource {

    var photos = [Photo]()

}
```

Collection View Data Source (continued)

- To conform to the **`UICollectionViewDataSource`** protocol, a type also needs to conform to the **`NSObjectProtocol`** .

- The easiest and most common way to conform to this protocol is to subclass from **`NSObject.`**

- The **`UICollectionViewDataSource`** protocol declares two required methods to implement:

```
func collectionView(_ collectionView: UICollectionView,
                    numberOfItemsInSection section: Int) -> Int
func collectionView(_ collectionView: UICollectionView,
                    cellForItemAt indexPath: IndexPath) -> UICollectionViewCell
```

Collection View Data Source (continued)

- The first data source callback asks **how many cells** to display, and the second asks for the **`UICollectionViewCell`** to display for a given index path.

- Implement these two methods in **`PhotoDataSource.swift`**.

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

func collectionView(_ collectionView: UICollectionView,
                    cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

    let identifier = "UICollectionViewCell"
    let cell =
        collectionView.dequeueReusableCell(withReuseIdentifier: identifier,
                                           for: indexPath)

    return cell
}
```

Collection View Data Source (continued)

- Next, the collection view needs to know that an instance of **PhotoDataSource** is the data source object.

- In **PhotosViewController.swift** , add a property to reference an instance of **PhotoDataSource** and an outlet for a **UICollectionView** instance.

- Also, you will not need the **imageView** anymore, so delete it.

```
class PhotosViewController: UIViewController {

    @IBOutlet var imageView: UIImageView!
    @IBOutlet var collectionView: UICollectionView!

    var store: PhotoStore!
    let photoDataSource = PhotoDataSource()
```

Collection View Data Source (continued)

- Without the **imageView** property, you will not need the method **`updateImageView(for:)`** anymore. Go ahead and remove it.

```
func updateImageView(for photo: Photo) {
    store.fetchImage(for: photo) {
        (imageResult) -> Void in

        switch imageResult {
        case let .success(image):
            self.imageView.image = image
        case let .failure(error):
            print("Error downloading image: \(error)")
        }
    }
}
```

Collection View Data Source (continued)

- Update **`viewDidLoad()`** to set the data source on the collection view.

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

    collectionView.dataSource = photoDataSource
```

Collection View Data Source (continued)

- Finally, update the **`photoDataSource`** object with the result of the web service request and reload the collection view.

CENTENNIAL COLLEGE

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

    collectionView.dataSource = photoDataSource

    store.fetchInterestingPhotos {
        (photosResult) -> Void in

        switch photosResult {
        case let .success(photos):
            print("Successfully found \(photos.count) photos.")
            if let firstPhoto = photos.first {
                self.updateImageView(for: firstPhoto)
            }
            self.photoDataSource.photos = photos
        case let .failure(error):
            print("Error fetching interesting photos: \(error)")
            self.photoDataSource.photos.removeAll()
        }
        self.collectionView.reloadSections(IndexSet(integer: 0))
    }
}
```

Collection View Data Source (continued)

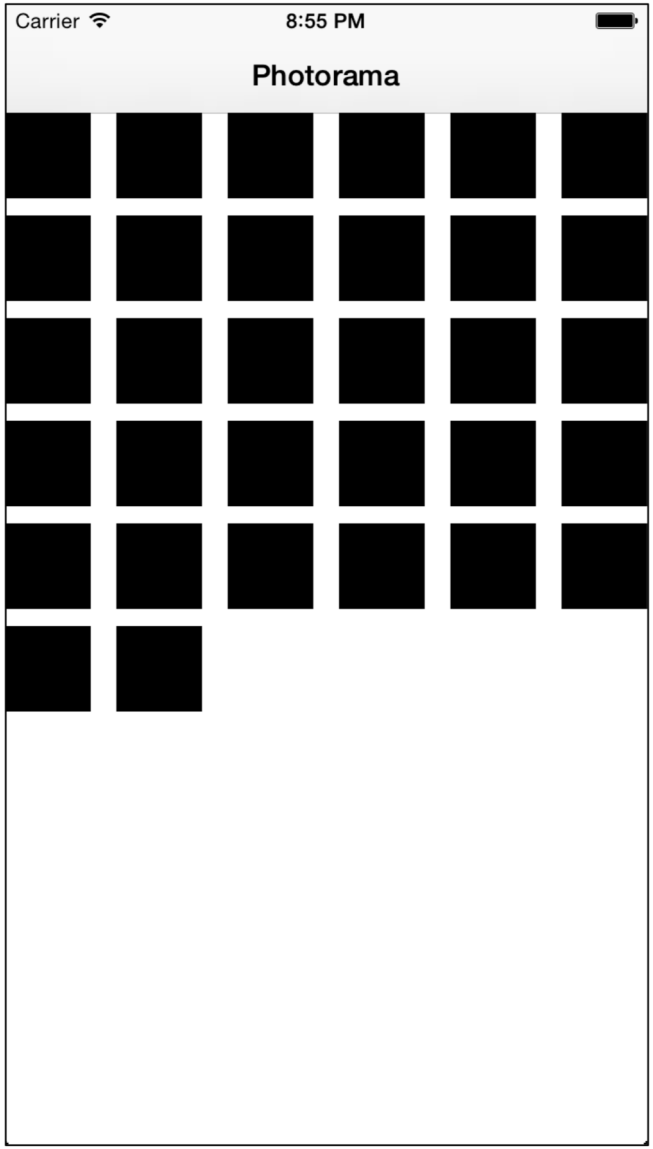
- The last thing you need to do is make the **collectionView** outlet connection.

- Open **Main.storyboard** and navigate to the collection view.

- Control-drag from the **Photorama** view controller to the **collection view** and connect it to the **collectionView** outlet.

Collection View Data Source (continued)

- Build and run the application. After the web service request completes, check the console to confirm that photos were found.
- On the iOS device, there will be a grid of black squares corresponding to the number of photos found.
- These cells are arranged in a flow layout . A flow layout fits as many cells on a row as possible before flowing down to the next row.
- If you rotate the iOS device, you will see the cells fill the given area.



Customizing the Layout

- The display of cells is not driven by the collection view itself but by the collection view's layout .

- The layout object is responsible for the placement of cells onscreen. Layouts, in turn, are driven by a subclass of **`UICollectionViewLayout`** .

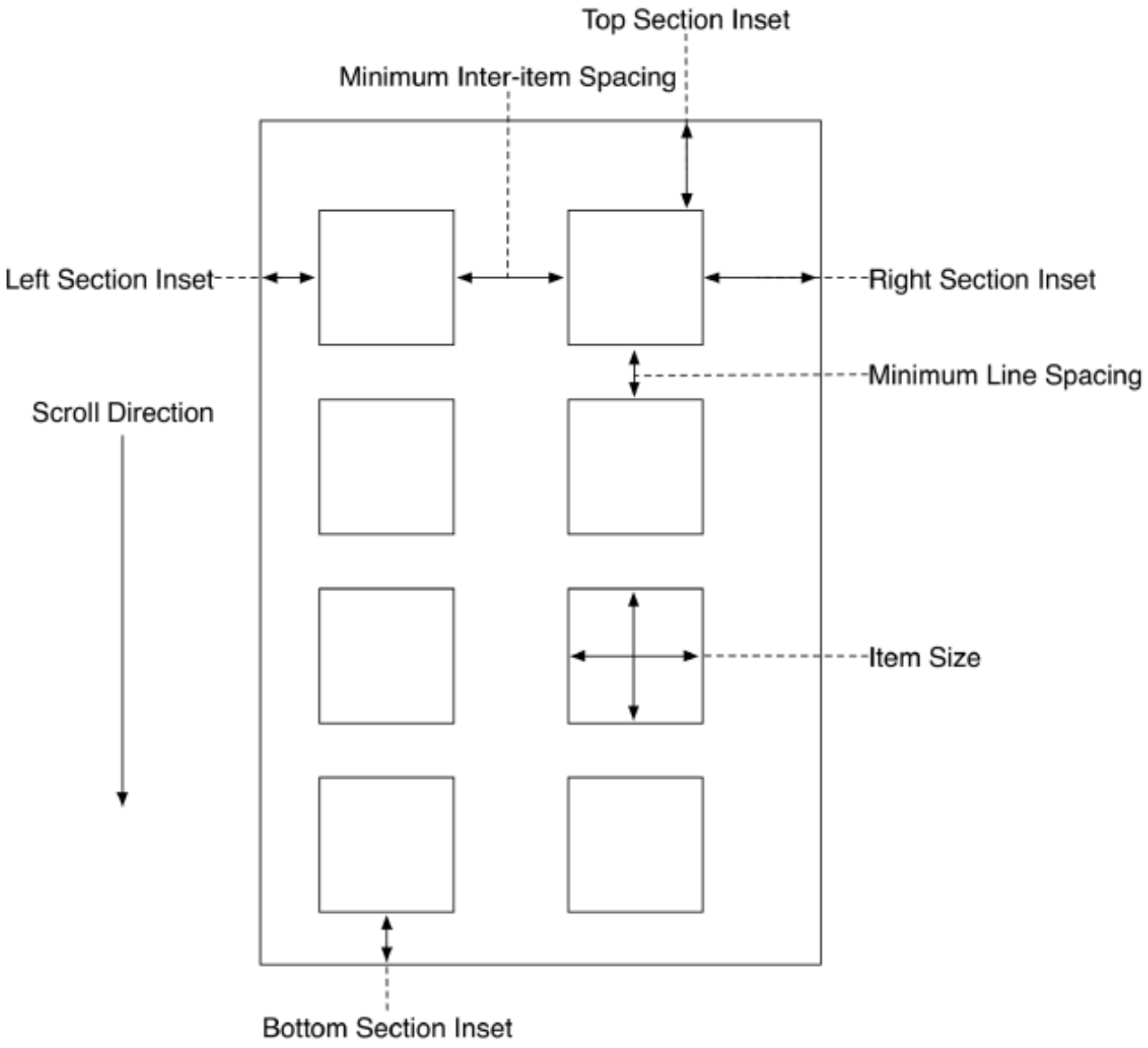
- The flow layout that Photorama is currently using is UICollectionViewFlowLayout , which is the only concrete **`UICollectionViewLayout`** subclass provided by the **`UIKit`** framework.

Customizing the Layout (continued)

- Some of the properties you can customize on UICollectionViewFlowLayout are:
  - **`scrollDirection`** – Do you want to scroll vertically or horizontally?
  - **`minimumLineSpacing`** – What is the minimum spacing between lines?
  - **`minimumInteritemSpacing`** – What is the minimum spacing between items in a row (or column, if scrolling horizontally)?
  - **`itemSize`** – What is the size of each item?
  - **`sectionInset`** – What are the margins used to lay out content for each section?

Customizing the Layout (continued)

CENTENNIAL COLLEGE

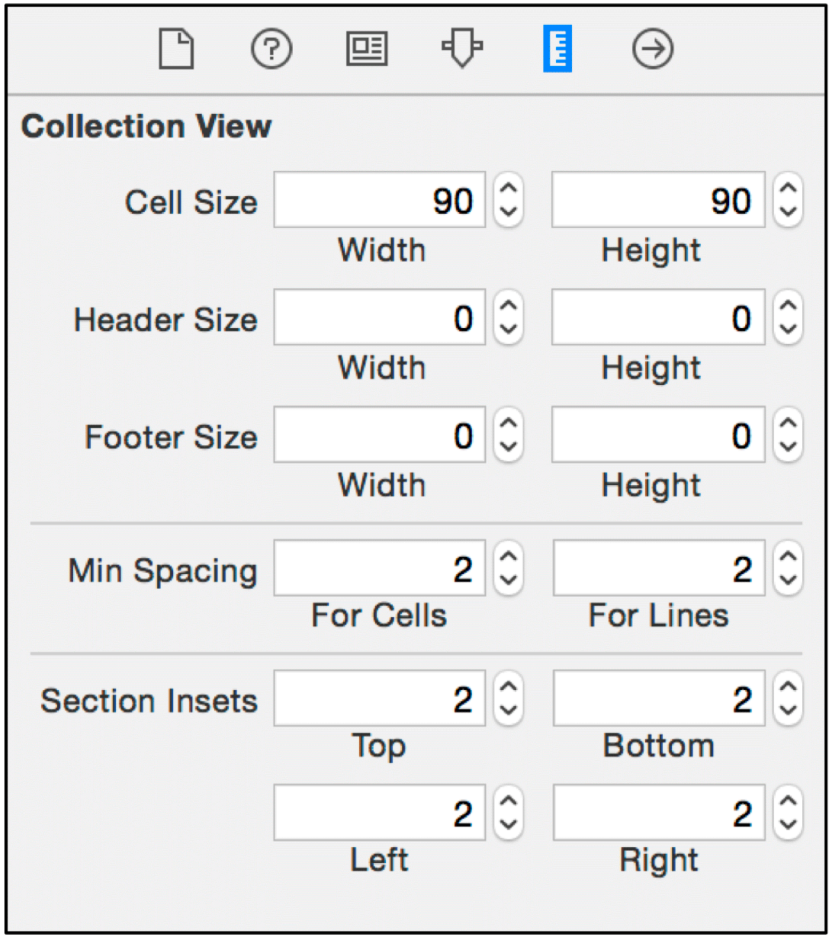


Customizing the Layout (continued)

CENTENNIAL COLLEGE

- Open **`Main.storyboard`** and select the collection view. Open the **size inspector** and configure the **Cell Size** , **Min Spacing** , and **Section Insets** as shown in the following Figure

- Build and run the application to see how the layout has changed



Creating a Custom UICollectionViewCell

- Next you are going to create a custom **`UICollectionViewCell`** subclass to display the photos.

- While the image data is downloading, the collection view cell will display a spinning activity indicator using the **`UIActivityIndicatorView`** class.

Creating a Custom UICollectionViewCell (continued)

- Create a new Swift file named **`PhotoCollectionViewCell`** and define **`PhotoCollectionViewCell`** as a subclass of **`UICollectionViewCell`**.

- Then add **outlets** to reference the image view and the activity indicator view.

```
import UIKit

class PhotoCollectionViewCell: UICollectionViewCell {

    @IBOutlet var imageView: UIImageView!
    @IBOutlet var spinner: UIActivityIndicatorView!

}
```

Creating a Custom UICollectionViewCell (continued)

- The activity indicator view should only spin when the cell is not displaying an image. Instead of always updating the spinner when the imageView is updated, or vice versa, you will write a helper method to take care of it for you.

- Create this helper method in **PhotoCollectionViewCell.swift**.

```
func update(with image: UIImage?) {
    if let imageToDisplay = image {
        spinner.stopAnimating()
        imageView.image = imageToDisplay
    } else {
        spinner.startAnimating()
        imageView.image = nil
    }
}
```

Creating a Custom UICollectionViewCell (continued)

- It would be nice to reset each cell to the spinning state both when the cell is first created and when the cell is getting reused.

- The method **`awakeFromNib()`** will be used for the former, and the method **`prepareForReuse()`** will be used for the latter.

- The method **`prepareForReuse()`** is called when a cell is about to be reused.

Creating a Custom UICollectionViewCell (continued)

- Implement these two methods in **`PhotoCollectionViewCell.swift`** to reset the cell back to the spinning state.

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

    update(with: nil)
}

override func prepareForReuse() {
    super.prepareForReuse()

    update(with: nil)
}
```

Creating a Custom UICollectionViewCell (continued)

- You will use a prototype cell to set up the interface for the collection view cell in the storyboard.

- If you recall, each **prototype cell** corresponds to a visually unique cell with a unique reuse identifier.

- Most of the time, the prototype cells will be associated with different `UICollectionViewCell` subclasses to provide behavior specific to that kind of cell.

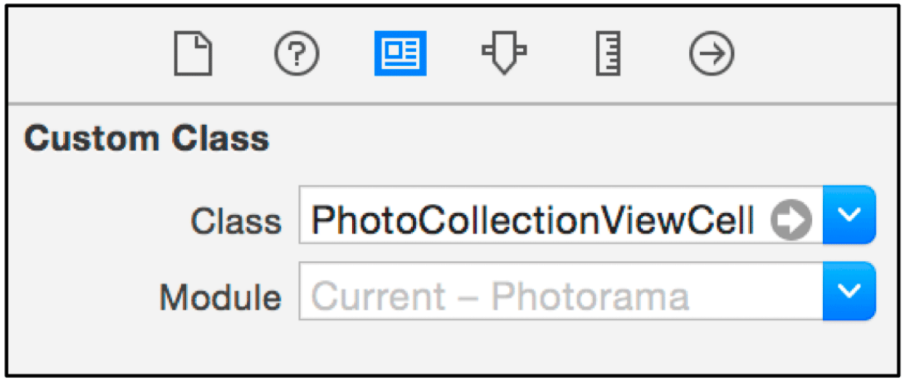
- In the collection view's attributes inspector, you can adjust the number of Items that the collection view displays, and each item corresponds to a prototype cell in the canvas.

- For **Photorama** , you only need one kind of cell: the **PhotoCollectionViewCell** that displays a photo.

Creating a Custom UICollectionViewCell (continued)

- Open **Main.storyboard** and select the collection view cell.

- In the identity inspector, change the Class to **PhotoCollectionViewCell** and, in the attributes inspector, change the Identifier to **PhotoCollectionViewCell** .

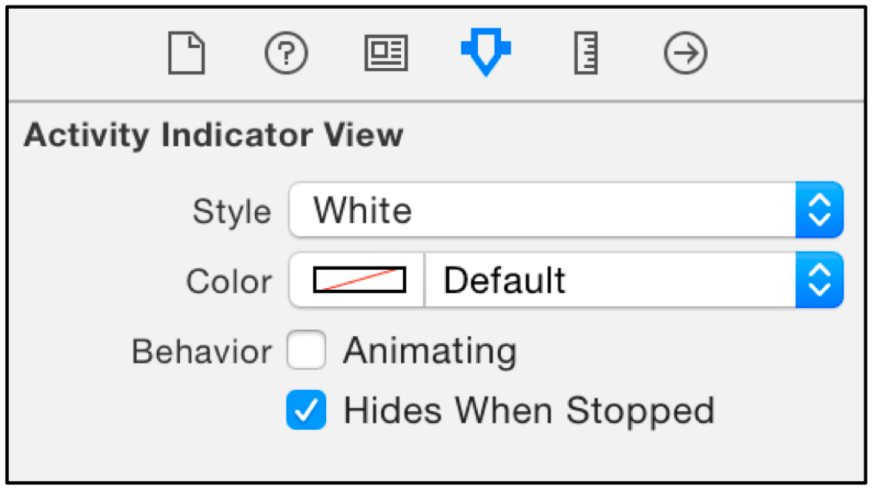


Creating a Custom UICollectionViewCell (continued)

- Drag an **image view** onto the `UICollectionViewCell`.
- Add **constraints** to pin the image view to the edges of the cell.
- Open the attributes inspector for the image view and set the Content Mode to **Aspect Fill** .
- This will cut off parts of the photos, but it will allow the photos to completely fill in the collection view cell.

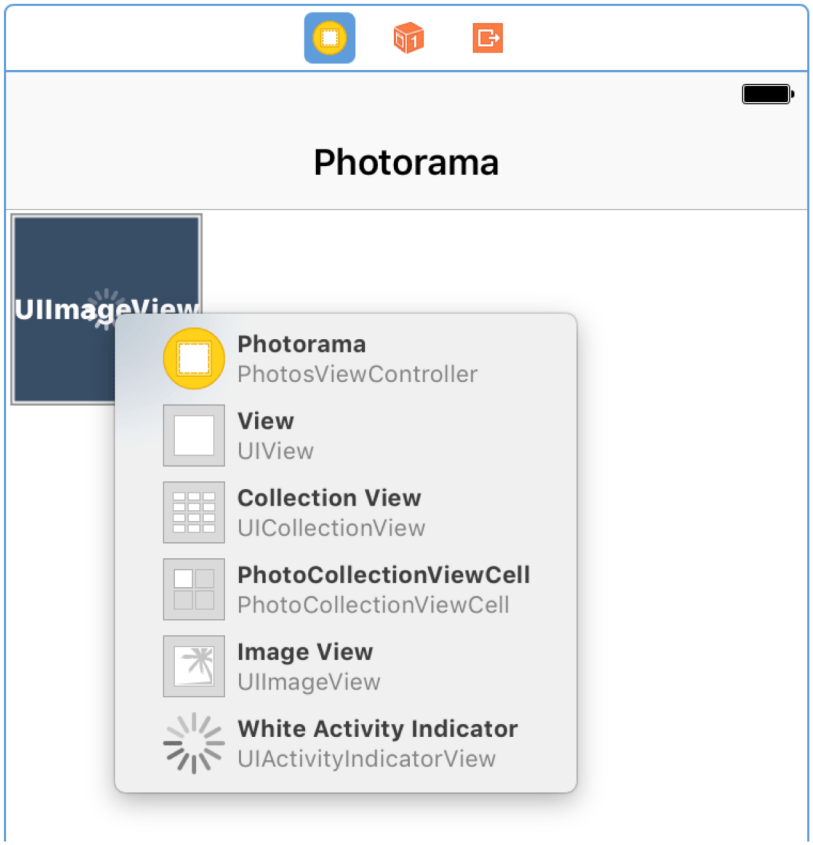
Creating a Custom UICollectionViewCell (continued)

- Next, drag an **activity indicator view** on top of the **image view.**
- Add constraints to center the activity indicator view both **horizontally** and **vertically** with the image view.
- Open its attributes inspector and select **Hides When Stopped**



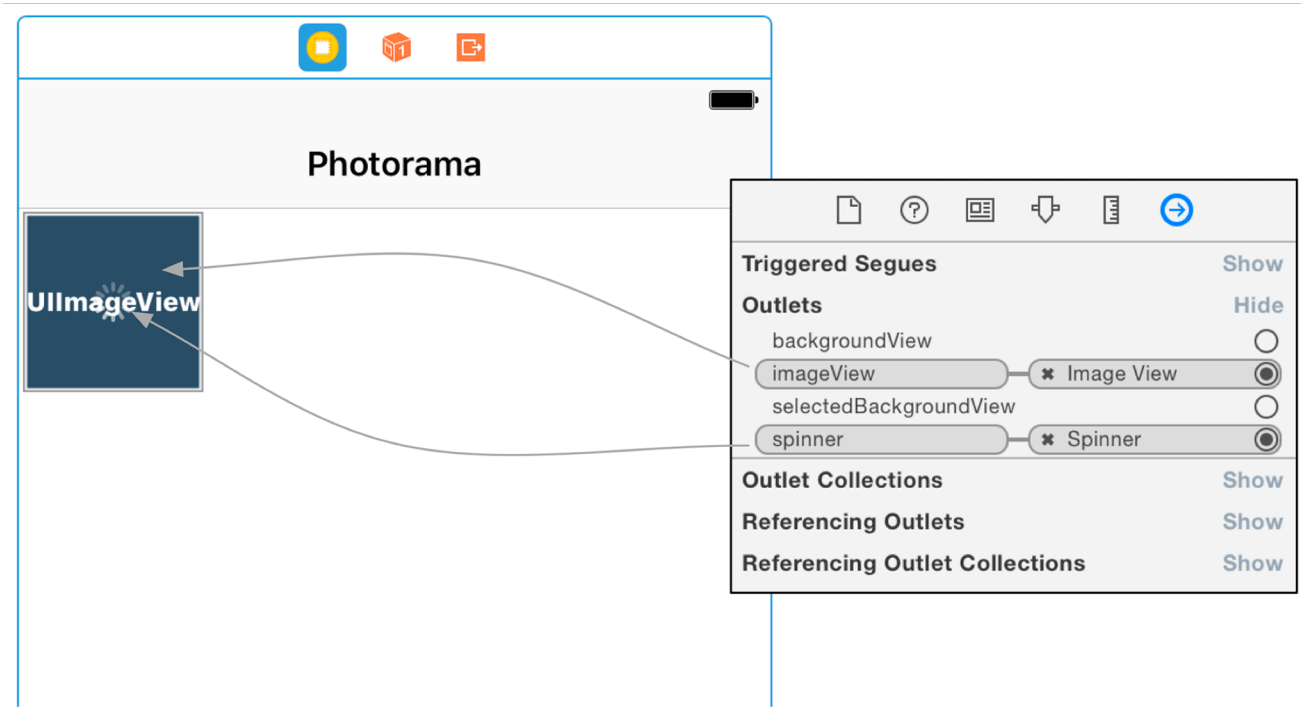
Creating a Custom UICollectionViewCell (continued)

- Select the **collection view cell** again.
- This can be a bit tricky to do on the canvas because the newly added subviews completely cover the cell itself.
- A helpful Interface Builder tip is to hold **Control** and **Shift** together and then click on top of the view you want to select.
- You will be presented with a list of all of the views and controllers under the point you clicked



Creating a Custom UICollectionViewCell (continued)

- With the cell selected, open the **connections** inspector and connect the **imageView** and **spinner** properties to the image view and activity indicator view on the canvas



Creating a Custom UICollectionViewCell (continued)

- Next, open **PhotoDataSource.swift** and update the data source method to use the **PhotoCollectionViewCell** .

CENTENNIAL COLLEGE

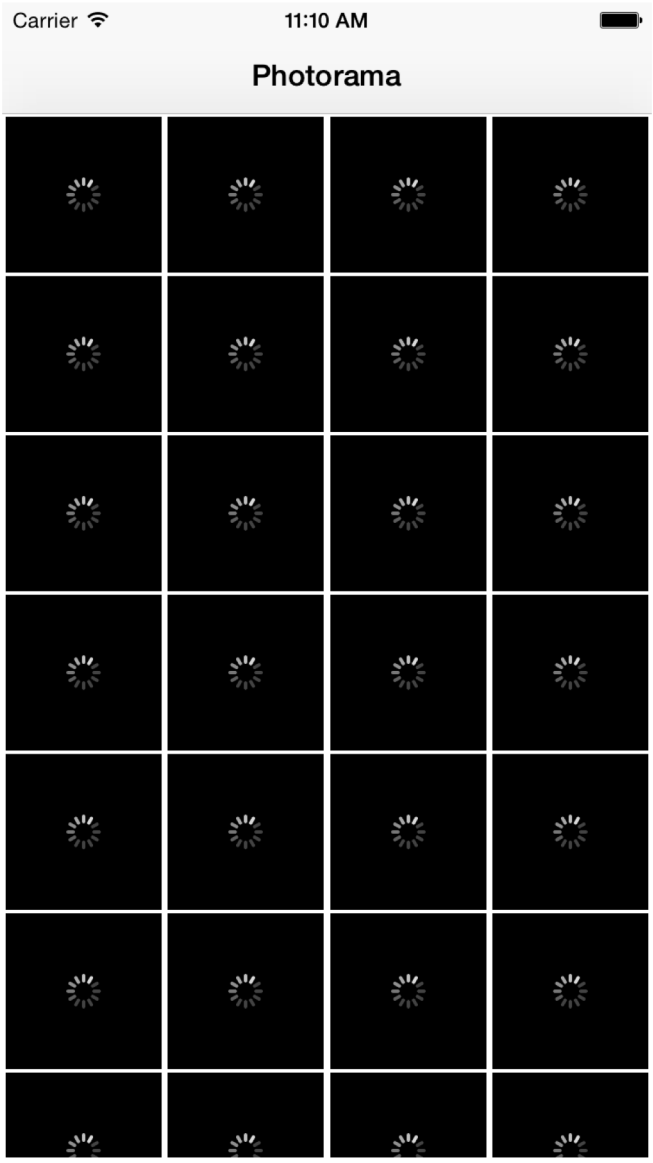
```
func collectionView(_ collectionView: UICollectionView,
                    cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

    let identifier = "UICollectionViewCell" "PhotoCollectionViewCell"
    let cell =
        collectionView.dequeueReusableCell(withReuseIdentifier: identifier,
                                           for: indexPath) as! PhotoCollectionViewCell

    return cell
}
```

Creating a Custom UICollectionViewCell (continued)

- Build and Run the application.



CENTENNIAL COLLEGE

Downloading the Image Data

- In **`PhotosViewController.swift`** , have the class conform to the **`UICollectionViewDelegate`** protocol.

```
class PhotosViewController: UIViewController, UICollectionViewDelegate {
```

- (Because the **`UICollectionViewDelegate`** protocol only defines optional methods, Xcode does not report any errors when you add this declaration.)

- Update **`viewDidLoad()`** to set the **PhotosViewController** as the delegate of the collection view.

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

    collectionView.dataSource = photoDataSource
    collectionView.delegate = self
```

Downloading the Image Data (continued)

- Finally, implement the **delegate** method in **PhotosViewController.swift** .

```
func collectionView(_ collectionView: UICollectionView,
                    willDisplay cell: UICollectionViewCell,
                    forItemAt indexPath: IndexPath) {

    let photo = photoDataSource.photos[indexPath.row]

    // Download the image data, which could take some time
    store.fetchImage(for: photo) { (result) -> Void in

        // The index path for the photo might have changed between the
        // time the request started and finished, so find the most
        // recent index path

        // (Note: You will have an error on the next line; you will fix it soon)
        guard let photoIndex = self.photoDataSource.photos.index(of: photo),
            case let .success(image) = result else {
                return
        }
        let photoIndexPath = IndexPath(item: photoIndex, section: 0)

        // When the request finishes, only update the cell if it's still visible
        if let cell = self.collectionView.cellForItem(at: photoIndexPath)
                                                as? PhotoCollectionViewCell {
            cell.update(with: image)
        }
    }
}
```

Downloading the Image Data (continued)

- In **Photo.swift** , declare that Photo conforms to the Equatable protocol and implement the required overloading of the **==** operator.

```
class Photo: Equatable {
    ...
    static func == (lhs: Photo, rhs: Photo) -> Bool {
        // Two Photos are the same if they have the same photoID
        return lhs.photoID == rhs.photoID
    }
}

extension Photo: Equatable {
    static func == (lhs: Photo, rhs: Photo) -> Bool {
        // Two Photos are the same if they have the same photoID
        return lhs.photoID == rhs.photoID
    }
}
```