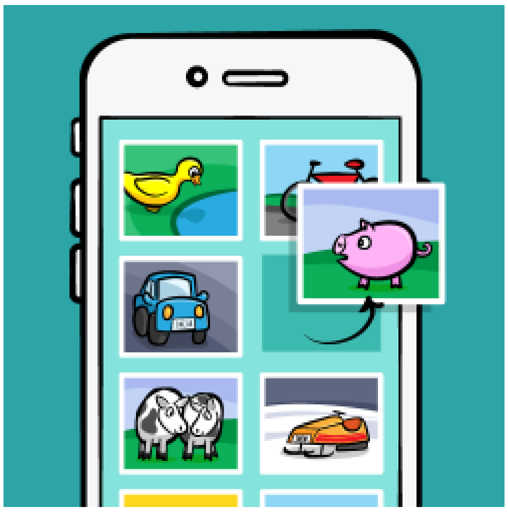
UICollectionView Tutorial: Reusable Views, Selection, and Reordering

Bradley Johnson on September 5, 2016

Note: This tutorial has been updated for Swift 3, iOS 10, and Xcode 8.

In the first part of this tutorial, you saw how to use a `UICollectionView` to display a grid of photos.

In this second and final part of the tutorial, you will continue the journey and learn how to interact with a collection view as well as customize it a bit further with headers. You'll continue working where you left off in part 1 so open your project or FlickrSearch from part 1 and start from there (though you'll still need to get a **new API key** as shown in part 1. If it's been a while since you did part 1, you will also need a new API key).



Create your own grid-based photo browsing app with collection views!

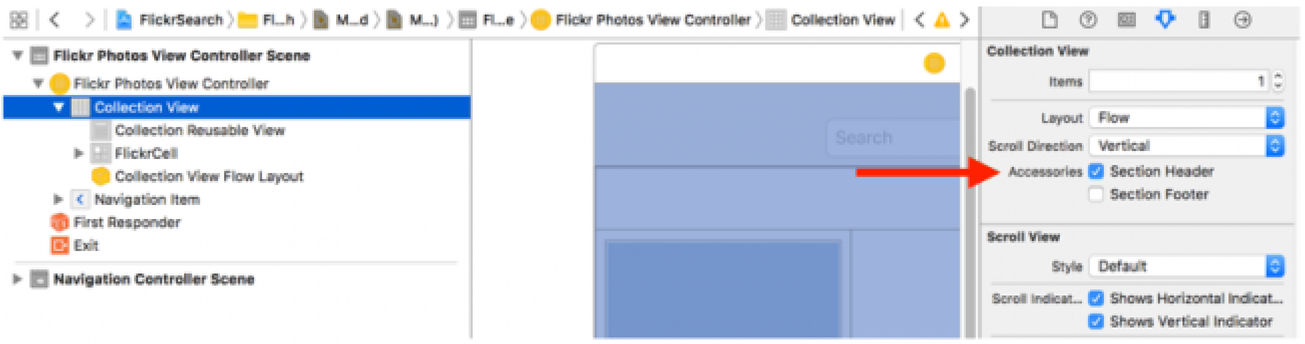
Adding a header

The app has one section per set of search results. It would be nice to add a header before each set of search results, to give the user a bit more context about the photos.

You will create this header using a class called `UICollectionReusableView`. This class is kind of like a collection view cell (in fact, cells inherit from this class), but used for other things like headers or footers.

This view can be built inside of your storyboard and connected to its own class. Choose **File\New\File…**, choose the **iOS\Source\Cocoa Touch Class** template and click **Next**. Name the class **FlickrPhotoHeaderView** making it a subclass of **UICollectionReusableView**. Click **Next** and then **Create** to save the file.

Open **MainStoryboard.storyboard** and click on the collection view inside of the Scene Inspector on the left (you might need to drill down a couple of levels from the main view first). Open the Attributes Inspector and check the **Section Header** box under Accessories:



FlickrSearch 〉 Fl...h 〉 M...d 〉 M...) 〉 Fl...e 〉 Flickr Photos View Controller 〉 Collection View

- Flickr Photos View Controller Scene
  - Flickr Photos View Controller
    - Collection View
      - Collection Reusable View
      - FlickrCell
      - Collection View Flow Layout
    - Navigation Item
    - First Responder
    - Exit
- Navigation Controller Scene

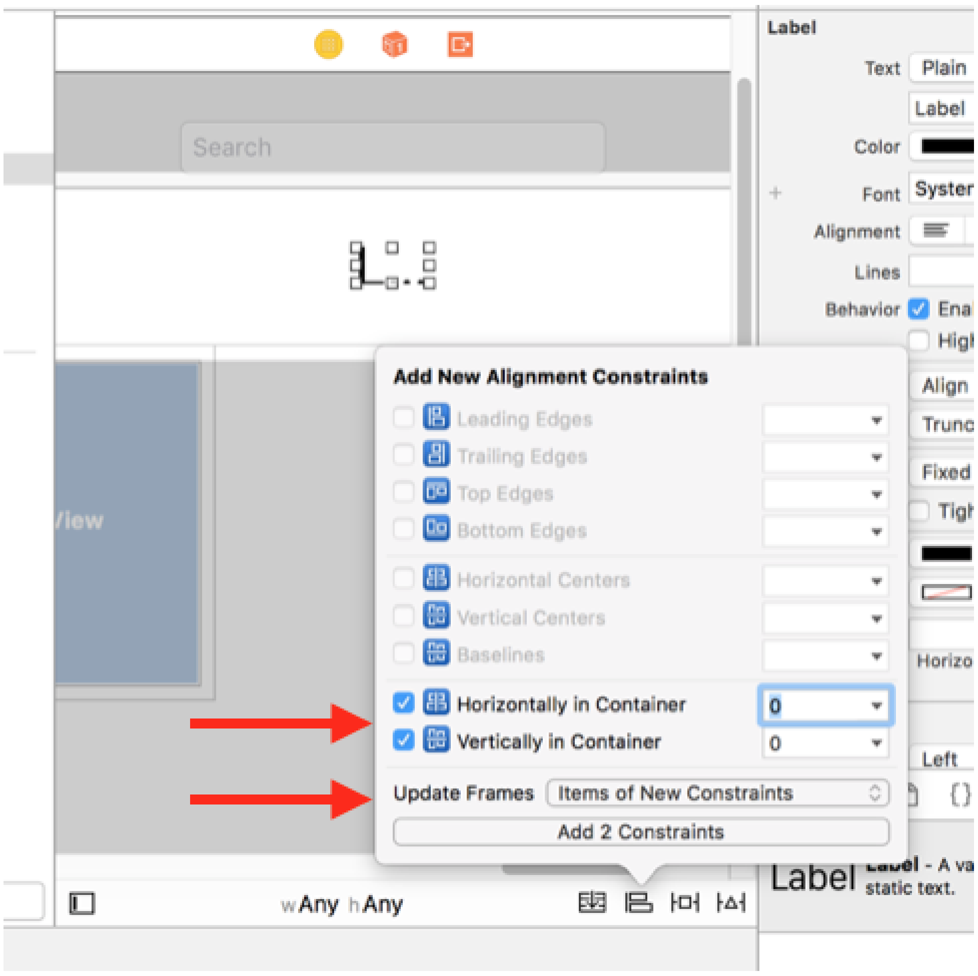
Collection View
Items: 1
Layout: Flow
Scroll Direction: Vertical
Accessories: [✓] Section Header
[ ] Section Footer

Scroll View
Style: Default
Scroll Indicat...: [✓] Shows Horizontal Indicat...
[✓] Shows Vertical Indicator

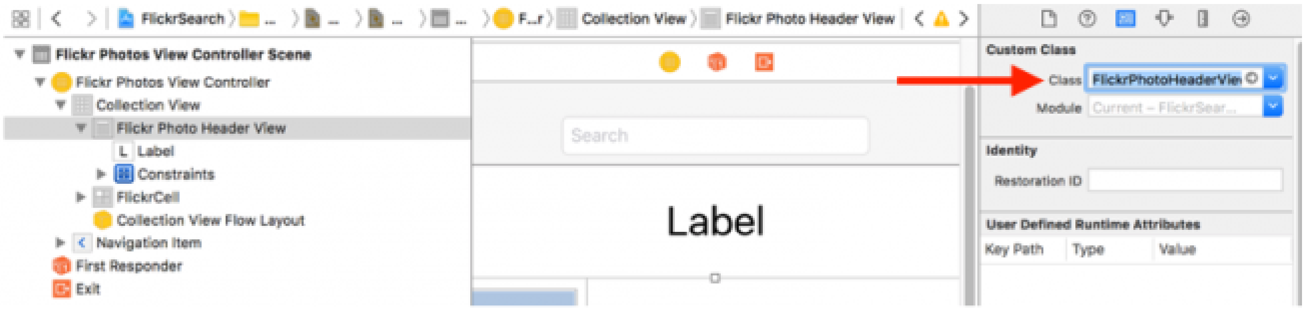
If you look at the scene inspector on the left, a "Collection Reusable View" has automatically been added under the Collection View. Click on the **Collection Reusable View** to select it, and you can begin adding subviews.

To give you a little more space to work with, click the white handle at the bottom of the view and drag it down, making the view 90 pixels tall. (Or, you can set the size for the view explicitly via the Size Inspector.)

Drag a label into the header view and center it using the guides. Change its **Font** to System 32.0, then go to the alignment menu and pin it to the horizontal and vertical centers of the container, and update the frame:



Select the header view itself, open the identity inspector and set the **Class** to **FlickrPhotoHeaderView**:


Flickr Photos View Controller Scene
Flickr Photos View Controller
Collection View
Flickr Photo Header View
Label
Constraints
FlickrCell
Collection View Flow Layout
Navigation Item
First Responder
Exit
Search
Label
Custom Class
Class FlickrPhotoHeaderView
Module Current – FlickrSear...
Identity
Restoration ID
User Defined Runtime Attributes
Key Path Type Value

Open the Attributes Inspector and set the **Background** to **Group Table View Background Color**, which is a nice light grey, and set the Identifier to **FlickrPhotoHeaderView**. This is the identifier that will be used when dequeuing this view.

Open the Assistant editor, making sure **FlickrPhotoHeaderView.swift** is open, and control-drag from the label to the class to make a new outlet, naming it **label**:

```
class FlickrPhotoHeaderView: UICollectionReusableView {
  @IBOutlet weak var label: UILabel!
}
```

If you build and run the app at this point, you still won't see a header (even if it is just a blank one with the word "Label"). There's another datasource method you need to implement. Open **FlickrPhotosViewController.swift** and add the following method to the `UICollectionViewDataSource` extension below the `collectionView(_:numberOfItemsInSection:)` method:

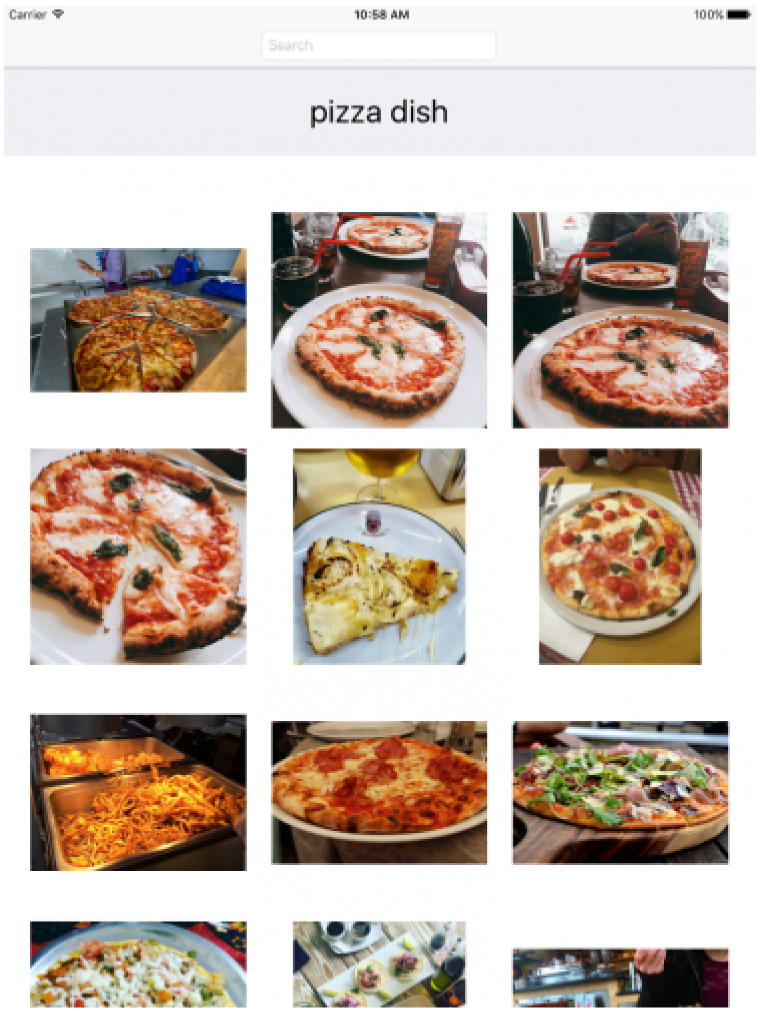
```
override func collectionView(_ collectionView: UICollectionView,
                             viewForSupplementaryElementOfKind kind: String,
                             at indexPath: IndexPath) -> UICollectionReusableView {
```

```
  //1
  switch kind {
  //2
  case UICollectionElementKindSectionHeader:
    //3
    let headerView = collectionView.dequeueReusableSupplementaryView(ofKind: kind,
                      withReuseIdentifier: "FlickrPhotoHeaderView",
                      for: indexPath) as! FlickrPhotoHeaderView
    headerView.label.text = searches[(indexPath as NSIndexPath).section].searchTerm
    return headerView
  default:
    //4
    assert(false, "Unexpected element kind")
  }
}
```

This method is similar to **`cellForItemAt`**, but for supplementary views. Here's a step-by-step explanation of the code:

1. The **`kind`** parameter is supplied by the layout object and indicates which sort of supplementary view is being asked for.
2. **`UICollectionElementKindSectionHeader`** is a supplementary view kind belonging to the flow layout. By checking that box in the storyboard to add a section header, you told the flow layout that it needs to start asking for these views. There is also a **`UICollectionElementKindSectionFooter`**, which you're not currently using. If you don't use the flow layout, you don't get header and footer views for free like this.
3. The header view is dequeued using the identifier added in the storyboard. This works just like cell dequeuing. The label's text is then set to the relevant search term.
4. An assert is placed here to make it clear to other developers (including future you!) that you're not expecting to be asked for anything other than a header view.

This is a good spot to build and run. You will see that your UI is mostly complete. If you do multiple searches, you'll get nice section headers dividing up your results. As a bonus, try rotating the device – notice how the layout, including the headers, adapts perfectly, without any extra work required :]



Mmmmmmmmmmm Pizza

Interacting With Cells

In this final section of the tutorial you will learn some ways to interact with collection view cells. You'll take three different approaches. The first will display a larger version of the image. The second will demonstrate how to support multiple selection in order to share images. And lastly, you will learn how to allow the user to rearrange the images by dragging them.

Single selection

Collection views can animate changes to their layout. Your first task is to show a larger version of a photo when it is tapped.

First, you need to add a property to keep track of the tapped cell. Open **FlickrPhotosViewController.swift** and add the following property definition below the `itemsPerRow` property:

```
//1
var largePhotoIndexPath: NSIndexPath? {
  didSet {
    //2
    var indexPaths = [IndexPath]()
    if let largePhotoIndexPath = largePhotoIndexPath {
      indexPaths.append(largePhotoIndexPath)
    }
    if let oldValue = oldValue {
      indexPaths.append(oldValue)
    }
    //3
    collectionView?.performBatchUpdates({
      self.collectionView?.reloadItems(at: indexPaths)
    }) { completed in
      //4
      if let largePhotoIndexPath = self.largePhotoIndexPath {
        self.collectionView?.scrollToItem(
          at: largePhotoIndexPath,
          at: .centeredVertically,
          animated: true)
      }
    }
  }
}
```

Here's the step by step breakdown:

1. `largePhotoIndexPath` is an optional that will hold the index path of the tapped photo, if there is one.
2. Whenever this property gets updated, the collection view needs to be updated. a `didSet` property observer is the safest place to manage this. There may be two cells that need reloading, if the user has tapped one cell then another, or just one if the user has tapped the first cell, then tapped it again to shrink.
3. `performBatchUpdates` will animate any changes to the collection view performed inside the block. You want it to reload the affected cells.
4. Once the animated update has finished, it's a nice touch to scroll the enlarged cell to the middle of the screen

What enlarged cell?, I hear you asking. You'll get to that in a minute!

Tapping a cell will make the collection view select it. You want to know a cell has been tapped, so you can set the `largeIndexPath` property, but you don't actually want to select it, because that might get confusing later on when you're doing multiple selection. `UICollectionViewDelegate` has you covered. The collection view asks its delegate if it's OK to select a specific cell. Still in **FlickrPhotosViewController.swift**, add the following extension below your `UICollectionViewDataSource` extension:

```
// MARK: - UICollectionViewDelegate
extension FlickrPhotosViewController {

  override func collectionView(_ collectionView: UICollectionView,
                               shouldSelectItemAt indexPath: IndexPath) -> Bool {

    largePhotoIndexPath = largePhotoIndexPath == indexPath ? nil : indexPath
    return false
  }
}
```

This method is pretty simple. If the tapped cell is already the large photo, set the **largePhotoIndexPath** property to **nil**, otherwise set it to the index path the user just tapped. This will then call the property observer you added earlier and cause the collection view to reload the affected cell(s).

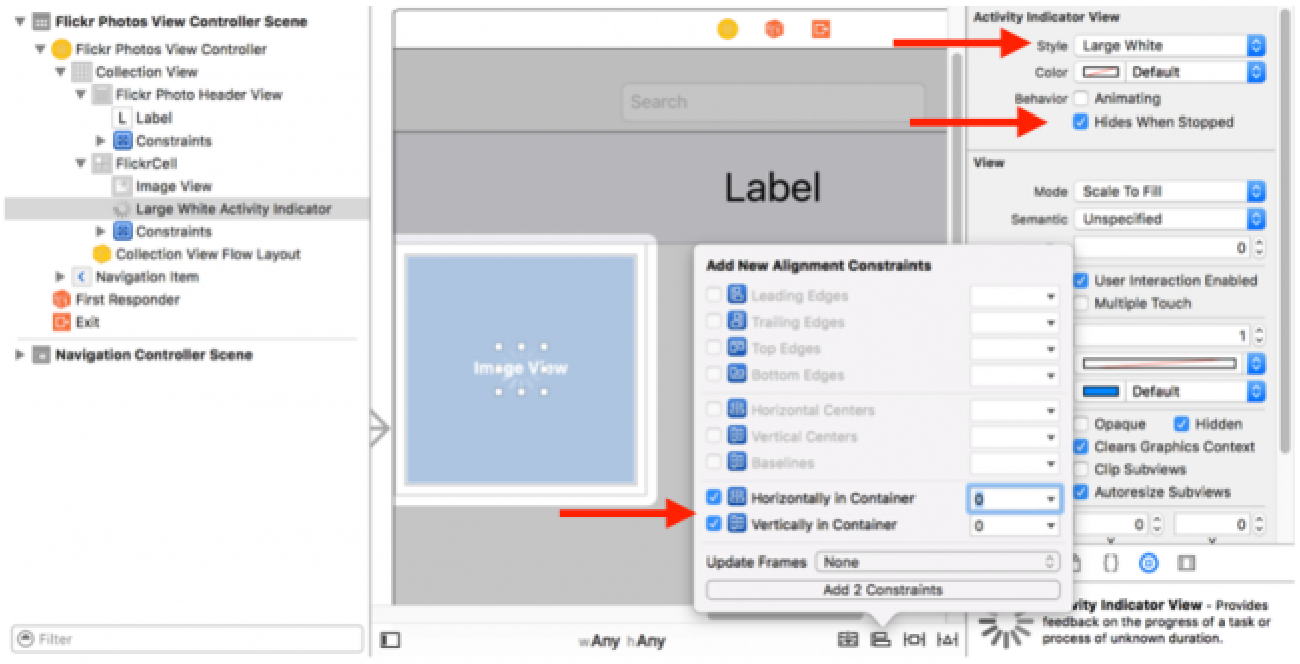
To make the tapped cell appear larger, you need to modify the **sizeForItemAt** flow layout delegate method. Add the following code to the beginning of the method:

```
// New code
if indexPath == largePhotoIndexPath {
  let flickrPhoto = photoForIndexPath(indexPath)
  var size = collectionView.bounds.size
  size.height -= topLayoutGuide.length
  size.height -= (sectionInsets.top + sectionInsets.right)
  size.width -= (sectionInsets.left + sectionInsets.right)
  return flickrPhoto.sizeToFillWidthOfSize(size)
}
```

This calculates the size of the cell to fill as much of the collection view as possible whilst maintaining its aspect ratio.

There's not much point in making a bigger cell unless you have a larger photo to show in it.

Open **Main.storyboard** and drag an activity indicator into the image view in the collection view cell. In the Attributes inspector, set the **Style** to **Large White** and check the **Hides When Stopped** box. Drag the indicator into the middle of cell (let the guides show you where that is) then, using the alignment menu, horizontally and vertically center the indicator in its container.



Style: Large White
Color: Default
Behavior: Animating ☐, Hides When Stopped ✓
Mode: Scale To Fill
Semantic: Unspecified
User Interaction Enabled ✓
Multiple Touch ☐
Opaque ☐, Hidden ✓
Clears Graphics Context ✓
Clip Subviews ☐
Autoresize Subviews ✓
Horizontally in Container ✓ 0
Vertically in Container ✓ 0
Update Frames: None

Open the assistant editor and control-drag from the activity indicator to **FlickrPhotoCell.swift** to add an outlet – call it **activityIndicator**:

```
@IBOutlet weak var activityIndicator: UIActivityIndicatorView!
```

Also in **FlickrPhotoCell.swift**, add the following code to give the cell control of its background color. You'll need this later on:

```
// MARK: - Properties
override var isSelected: Bool {
  didSet {
    imageView.layer.borderWidth = isSelected ? 10 : 0
  }
}

// MARK: - View Life Cycle
override func awakeFromNib() {
  super.awakeFromNib()
  imageView.layer.borderColor = themeColor.cgColor
  isSelected = false
}
```

Finally, open **FlickrPhotosViewController.swift** and replace the `collectionView(_:cellForItemAt:)` method with the following:

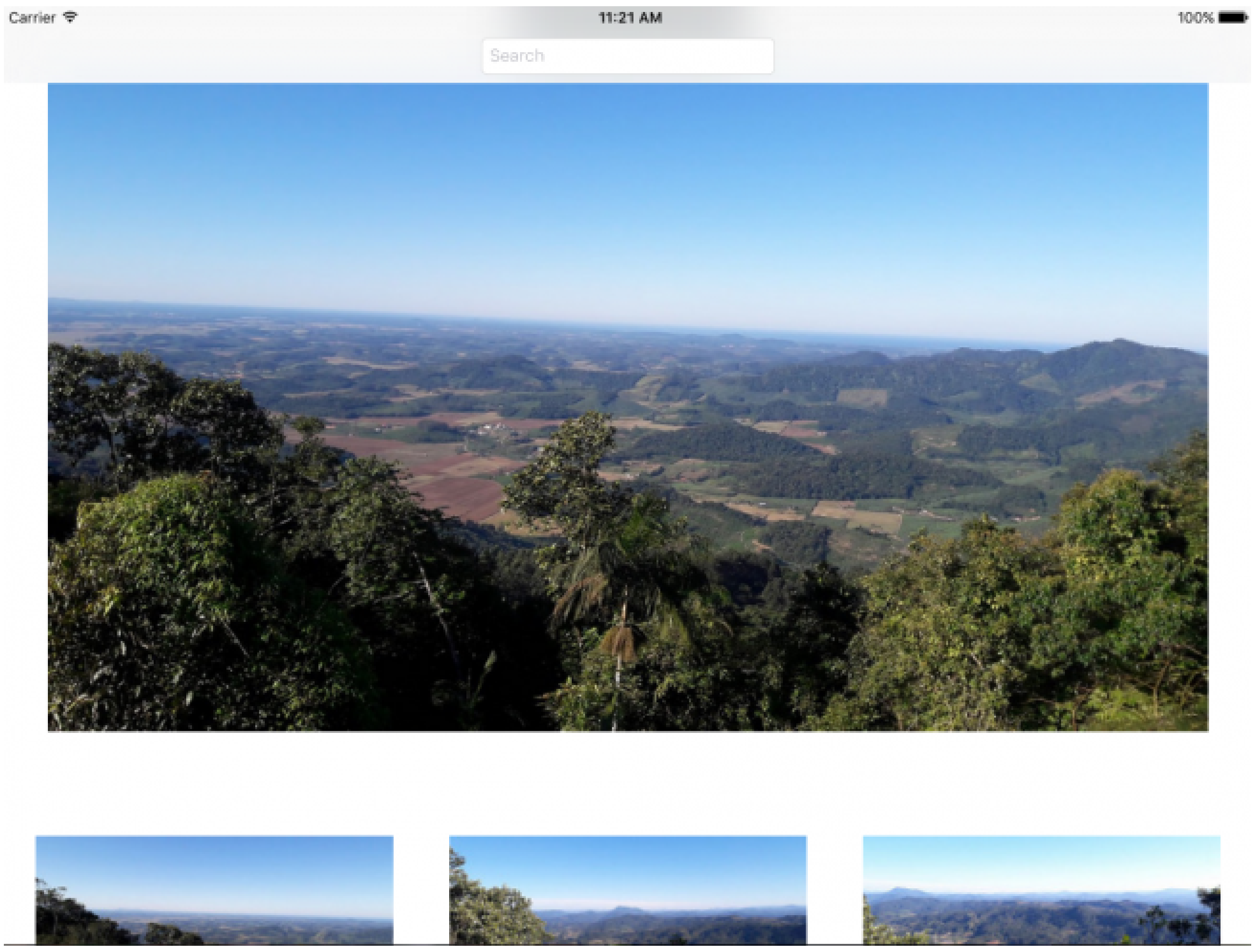
```
override func collectionView(_ collectionView: UICollectionView,
                             cellForItemAt indexPath: NSIndexPath) -> UICollectionViewCell
{

  let cell = collectionView.dequeueReusableCell(
             withReuseIdentifier: reuseIdentifier, for: indexPath) as! FlickrPhotoCell
  var flickrPhoto = photoForIndexPath(indexPath)

  //1
  cell.activityIndicator.stopAnimating()

  //2
  guard indexPath == largePhotoIndexPath else {
    cell.imageView.image = flickrPhoto.thumbnail
    return cell
  }

  //3
  guard flickrPhoto.largeImage == nil else {
    cell.imageView.image = flickrPhoto.largeImage
    return cell
  }

  //4
  cell.imageView.image = flickrPhoto.thumbnail
  cell.activityIndicator.startAnimating()

  //5
  flickrPhoto.loadLargeImage { loadedFlickrPhoto, error in

    //6
    cell.activityIndicator.stopAnimating()

    //7
    guard loadedFlickrPhoto.largeImage != nil && error == nil else {
      return
    }

    //8
    if let cell = collectionView.cellForItem(at: indexPath) as? FlickrPhotoCell,
      indexPath == self.largePhotoIndexPath  {
      cell.imageView.image = loadedFlickrPhoto.largeImage
    }
  }

  return cell
}
```

This is quite a long method now, so here's the step-by-step:

1. Always stop the activity spinner – you could be reusing a cell that was previously loading an image
2. This part is as before – if you're not looking at the large photo, just set the thumbnail and return

3. If the large image is already loaded, set it and return
4. By this point, you want the large image, but it doesn't exist yet. Set the thumbnail image and start the spinner going. The thumbnail will stretch until the download is complete
5. Ask for the large image from Flickr. This loads the image asynchronously and has a completion block
6. The load has finished, so stop the spinner
7. If there was an error or no photo was loaded, there's not much you can do.
8. Check that the large photo index path hasn't changed while the download was happening, and retrieve whatever cell is currently in use for that index path (it may not be the original cell, since scrolling could have happened) and set the large image.

Build and run, perform a search and tap a nice-looking photo – it grows to fill the screen, and the other cells move around to make space!



Tap the cell again, or try scrolling and tapping a different cell. You didn't have to write any code to move or animate those cells, the collection view and its layout object did all the hard work for you!

Multiple selection

Your next task for this tutorial is to let the user select multiple photos and share them with a friend. The process for multi-selection on a collection view is very similar to that of a table view. The only trick is to tell the collection view to allow multiple selection.

The process for selection works in the following way:

1. The user taps the Share button to tell the UICollectionView to allow multi- selection and set the sharing property to YES.
2. The user taps multiple photos that they want to share, adding them to an array.
3. The user taps the Share button again, which brings up the sharing interface.

4. When the user finishes sharing the images or taps Cancel, the photos are deselected and the collection view goes back to single selection mode.

First, open **FlickrPhotosViewController.swift** and add the following properties below the **`itemsPerRow`** property:

```
fileprivate var selectedPhotos = [FlickrPhoto]()
fileprivate let shareTextLabel = UILabel()
```

Next, add the following method to the **private** extension below the **`photoForIndexPath(_:)`** method:

```
func updateSharedPhotoCount() {
  shareTextLabel.textColor = themeColor
  shareTextLabel.text = "\(selectedPhotos.count) photos selected"
  shareTextLabel.sizeToFit()
}
```

The **`selectedPhotos`** array will keep track of the photos the user has selected, and the **`shareTextLabel`** will provide feedback to the user on how many photos have been selected. You will call **`updateSharedPhotoCount`** to keep **`shareTextLabel`** up to date.

Next, add the property following property below the **`largePhotoIndexPath`** property:

```
var sharing: Bool = false {
  didSet {
    collectionView?.allowsMultipleSelection = sharing
    collectionView?.selectItem(at: nil, animated: true, scrollPosition:
UICollectionViewScrollPosition())
    selectedPhotos.removeAll(keepingCapacity: false)

    guard let shareButton = self.navigationItem.rightBarButtonItems?.first else {
      return
    }

    guard sharing else {
      navigationItem.setRightBarButtonItems([shareButton], animated: true)
      return
    }

    if let _ = largePhotoIndexPath  {
      largePhotoIndexPath = nil
    }

    updateSharedPhotoCount()
    let sharingDetailItem = UIBarButtonItem(customView: shareTextLabel)
    navigationItem.setRightBarButtonItems([shareButton,sharingDetailItem], animated: true)
  }
}
```

`sharing` is a **`Bool`** with another property observer, similar to **`largePhotoIndexPath`** above. In this observer, you toggle the multiple selection status of the collection view, clear any existing selection and empty the selected photos array. You also update the bar button items to include and update the **`shareTextLabel`**.

Open **Main.storyboard** and drag a **UIBarButtonItem** to the **right** of the navigation bar above the collection view controller. In the **Attributes Inspector**, set the **System Item** to **Action** to give it the familiar sharing icon. Open the assistant editor, making sure **FlickrPhotosViewController.swift** is open, and control-drag from the bar button into the class to create a new action. Call the action **share:** and set the sender to be of type **UIBarButtonItem**.

Fill in the action method as shown:

```
@IBAction func share(_ sender: UIBarButtonItem) {
  guard !searches.isEmpty else {
    return
  }

  guard !selectedPhotos.isEmpty else {
    sharing = !sharing
    return
  }

  guard sharing else  {
    return
  }
  //TODO actually share photos!
}
```

At the moment, all this method does is some checking to make sure the user has actually searched for something, and has selected photos to share.

You actually want to allow the user to select cells now. Add the following code to the top of the `collectionView(_:shouldSelectItemAt:)` method:

```
guard !sharing else {
  return true
}
```

This will allow selection in sharing mode.

Next, add the following method to the `UICollectionViewDelegate` extension below the `collectionView(_:shouldSelectItemAth:)` method:

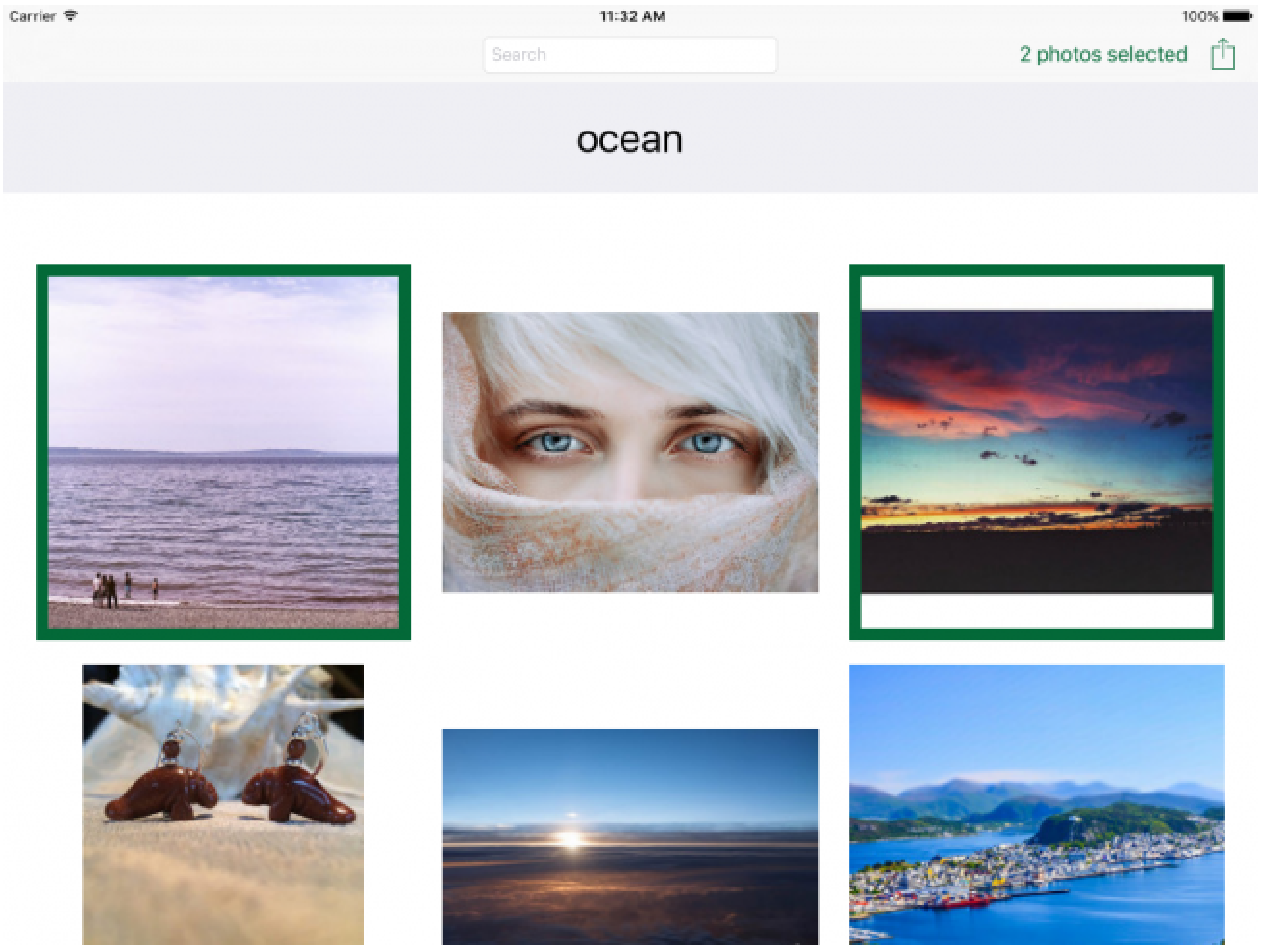
```
override func collectionView(_ collectionView: UICollectionView,
                             didSelectItemAt indexPath: IndexPath) {
  guard sharing else {
    return
  }

  let photo = photoForIndexPath(indexPath)
  selectedPhotos.append(photo)
  updateSharedPhotoCount()
}
```

This method allows adding selected photos to the shared photos array and updates the `shareTextLabel`. Finally, add the following method below the `collectionView(_:didSelectItemAtIndexPath:)` method:

```
override func collectionView(collectionView: UICollectionView,
                             didDeselectItemAtIndexPath indexPath: NSIndexPath) {

  guard sharing else {
    return
  }

  let photo = photoForIndexPath(indexPath)

  if let index = selectedPhotos.indexOf(photo) {
    selectedPhotos.removeAtIndex(index)
    updateSharedPhotoCount()
  }
}
```

This method removes a photo from the shared photos array and updates the `shareTextLabel`.

Build and run, and perform a search. Tap the share button to go into sharing mode and select different photos. The label will update and the selected cells will get a fetching Wenderlich Green border.



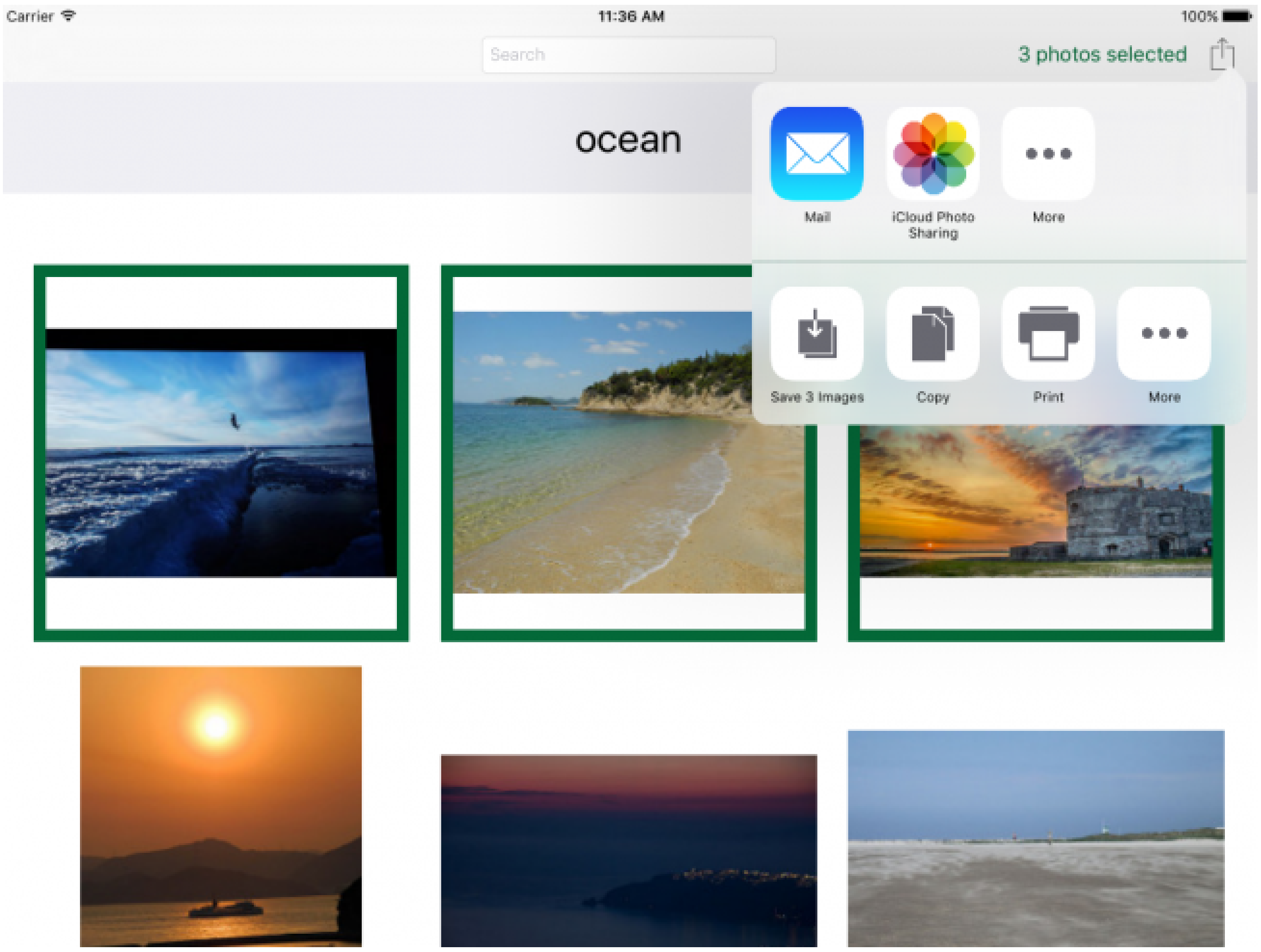
If you tap the share button again, everything just gets deselected, and you go back into non-sharing mode, where tapping a single photo enlarges it.

Of course, this share button isn't terribly useful unless there's actually a way to share the photos! Replace the `TODO` comment in your `share(_:)` method with the following code:

```
var imageArray = [UIImage]()
for selectedPhoto in selectedPhotos {
  if let thumbnail = selectedPhoto.thumbnail {
    imageArray.append(thumbnail)
  }
}

if !imageArray.isEmpty {
  let shareScreen = UIActivityViewController(activityItems: imageArray,
applicationActivities: nil)
  shareScreen.completionWithItemsHandler = { _ in
    self.sharing = false
  }
  let popoverPresentationController = shareScreen.popoverPresentationController
  popoverPresentationController?.barButtonItem = sender
  popoverPresentationController?.permittedArrowDirections = .any
  presentViewController(shareScreen, animated: true, completion: nil)
}
```

First, this code creates an array of `UIImage` objects from the `FlickrPhoto`'s thumbnails. The `UIImage` array is much more convenient, as we can simply pass it to a `UIActivityViewController`. The `UIActivityViewController` will show the user any image sharing services or actions available on the device: iMessage, Mail, Print, etc. You simply present your `UIActivityViewController` from within a popover (because this is an iPad app), and let the user take care of the rest!

Build and run, enter sharing mode, select some photos and hit the share button again. Your share dialog will appear!



Note: Testing exclusively on a simulator? You will find the simulator has far fewer sharing options than on device. If you are having trouble confirming that your share screen is properly sharing your images, try the **Save (x) Images** option. Whether on device or on simulator, this will save the selected images to Photos app, where you can review them to ensure everything worked.

Cell Reordering

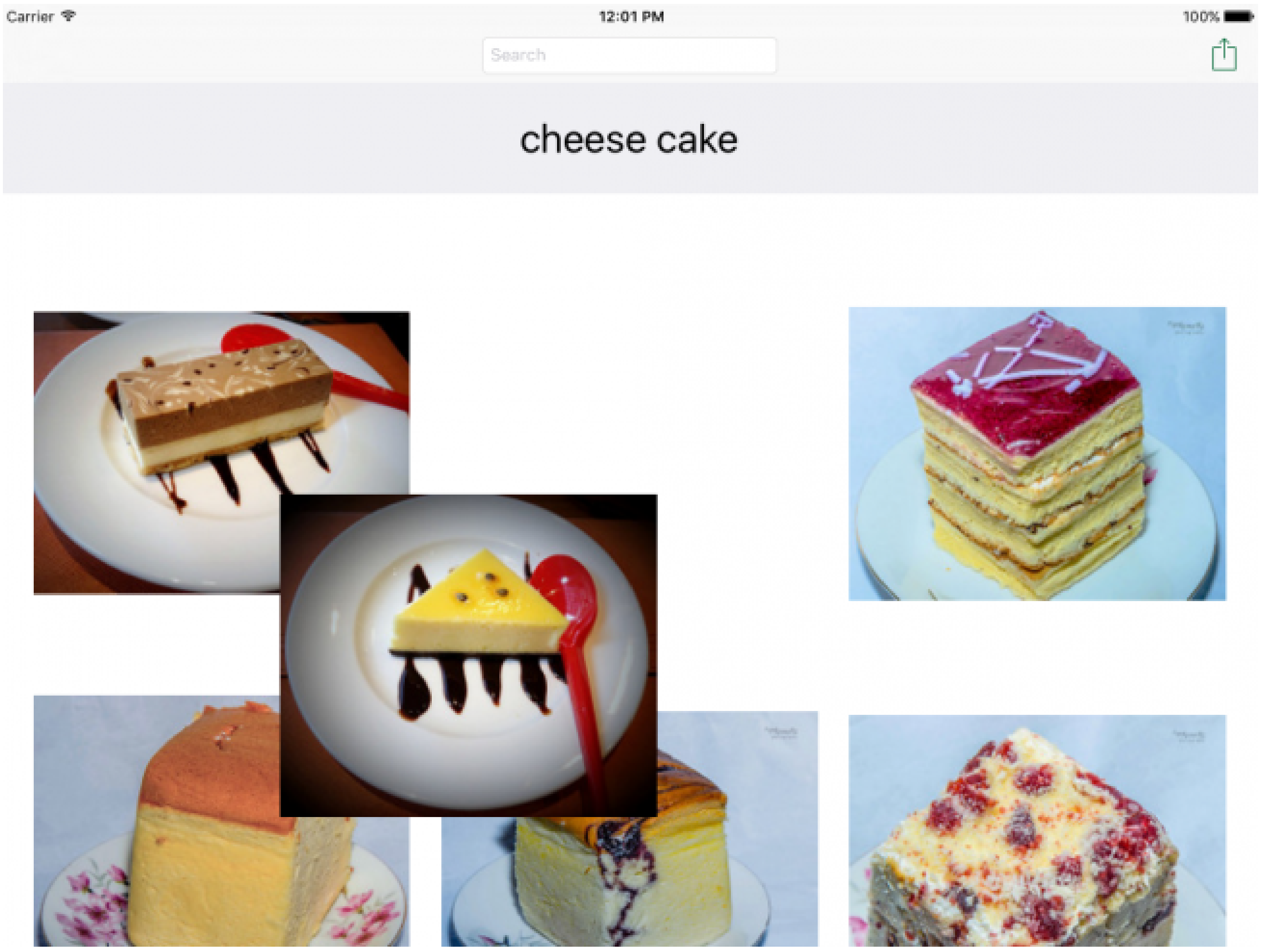
Prior to iOS 9, Collection View was sorely missing one feature that its cousin Table View has had for a long time: easy cell reordering. Thankfully, this is no longer the case, and it will be especially easy for you to implement it in this project because you are using a Collection View Controller.

Collection View Controllers now have a property called `installsStandardGestureForInteractiveMovement`, which by default is set to true. This property controls whether or not a gesture recognizer is installed onto the collection view to allow cell reordering. The only other thing you have to do to get cell reordering to work is override a single method from the **UICollectionViewDataSource** protocol. Open **FlickrPhotosViewController.swift**, add the following method below the `collectionView(_:cellForItemAt:)` method in your `UICollectionViewDataSource` extension:

```
override func collectionView(_ collectionView: UICollectionView,
                             moveItemAtIndexPath sourceIndexPath: IndexPath,
                             to destinationIndexPath: IndexPath) {

  var sourceResults = searches[(sourceIndexPath as NSIndexPath).section].searchResults
  let flickrPhoto = sourceResults.remove(at: (sourceIndexPath as NSIndexPath).row)

  var destinationResults = searches[(destinationIndexPath as
NSIndexPath).section].searchResults
  destinationResults.insert(flickrPhoto, at: (destinationIndexPath as NSIndexPath).row)
}
```

This is a pretty straight forward method to implement. You simply removed the moving item from it's spot in the results array, and then reinserted it into the array in its new position. That's it.

Give it a run, press/click and hold on an image and try moving it around to a new spot in the grid:



Not bad for 4 lines of code!

Where To Go From Here?

Here is the completed project that you developed in the tutorial series.

Congratulations, you have finished creating your very own stylish Flickr photo browser, complete with a cool UICollectionView based grid view!

In the process, you learned how to make custom UICollectionViewCells, create headers with `UICollectionReusableView`, detect when rows are tapped, implement multi-cell selection, and much more!

If you have any questions or comments about UICollectionViews or this tutorial, please join the forum discussion below!

Bradley Johnson

I'm a Mobile Developer at Getty Images Inc. in Seattle, Wa. I've been doing iOS for about 4 years now, and just started dabbling in Android as well. Swift is amazing! Go hawks.