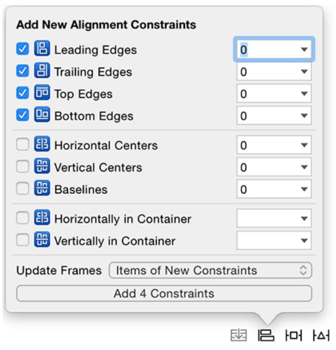
Collection Views-4

In this assignment you will continue working on the Soo Greyhounds app by displaying the latest photos in a grid using the UICollectionView class. This assignment will also reinforce the data source design pattern that you used in previous chapters.

1. Create a new repository on GitHub and add your instructors as collaborators.

2. Open your Soo Greyhounds project, then open the source control navigator. Remove your previous labs remote repository and add your new repository as a remote.

3. 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 image view on your PhotosViewController. 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.) Open the Auto Layout Align menu, configure it like this:



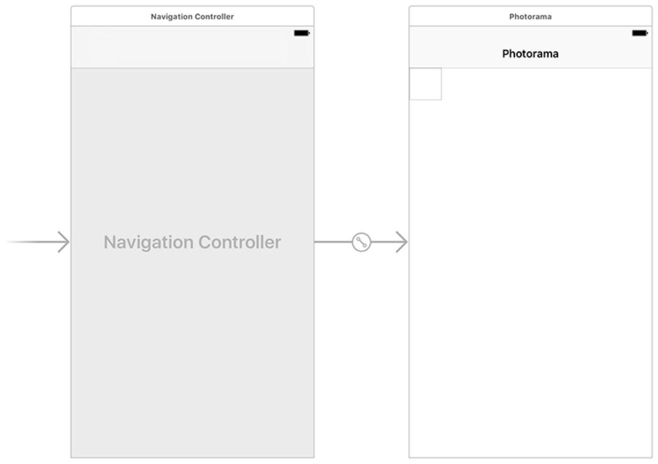
Add New Alignment Constraints

✓	Leading Edges	0
✓	Trailing Edges	0
✓	Top Edges	0
✓	Bottom Edges	0
☐	Horizontal Centers	0
☐	Vertical Centers	0
☐	Baselines	0
☐	Horizontally in Container	
☐	Vertically in Container	

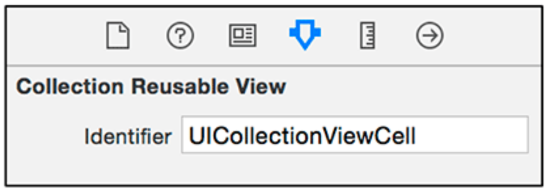
Update Frames: Items of New Constraints

Add 4 Constraints

4. 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 this:



5. 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:



Collection Reusable View

Identifier UICollectionViewCell

6. Create a new Swift file named PhotoDataSource and declare the PhotoDataSource class:

```
~~import Foundation~~
import UIKit

class PhotoDataSource: NSObject, UICollectionViewDataSource {
  var photos = [Photo]()
}
```

7. 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, as you did above. The UICollectionViewDataSource protocol declares two required methods to implement (numberOfItemsInSection and cellForItemAt). You might notice that these two methods look very similar to the two required methods of UITableViewDataSource. 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.

```
class PhotoDataSource: NSObject, UICollectionViewDataSource {
  var photos = [Photo]()

  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
  }
}
```

8. 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()
  …
```

9. 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)")~~
~~    }~~
~~  }~~
~~}~~
```

10. Update viewDidLoad() to set the data source on the collection view:

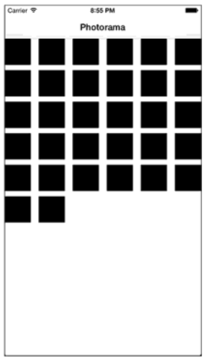
```
override func viewDidLoad() {
  super.viewDidLoad()

  collectionView.dataSource = photoDataSource
  …
```

11. Update the photoDataSource object with the result of the web service request and reload the collection view:

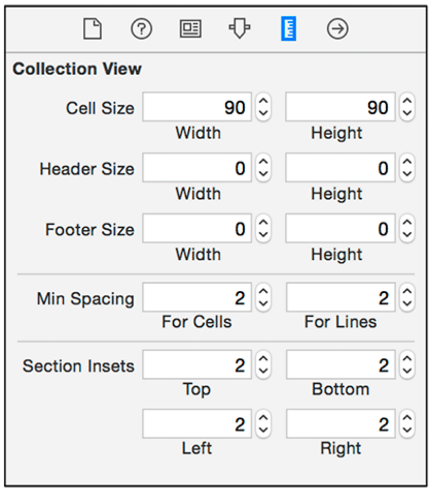
```
override func viewDidLoad()
   super.viewDidLoad()

   collectionView.dataSource = photoDataSource

   store.fetchLatestPhotos {
      (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 latest photos: \(error)")
         self.photoDataSource.photos.removeAll()
      }
      self.collectionView.reloadSections(IndexSet(integer: 0))
   }
}
```

12. Next you need to do is make the collectionView outlet connection. Open Main.storyboard and navigate to the collection view. Control-drag from the Soo Greyhounds view controller to the collection view and connect it to the collectionView outlet.

13. 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.



14. Open Main.storyboard and select the collection view. Open the size inspector and configure the Cell Size, Min Spacing, and Section Insets like this:



Collection View

Cell Size	90	90
	Width	Height
Header Size	0	0
	Width	Height
Footer Size	0	0
	Width	Height
Min Spacing	2	2
	For Cells	For Lines
Section Insets	2	2
	Top	Bottom
	2	2
	Left	Right

15. Build and run the application to see how the layout has changed.

16. 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. 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 Foundation~~
import UIKit
class PhotoCollectionViewCell: UICollectionViewCell {
  @IBOutlet var imageView: UIImageView!
  @IBOutlet var spinner: UIActivityIndicatorView!
}
```

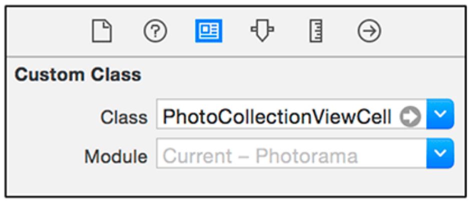
17. The activity indicator view should only spin when the cell is not displaying an image. Instead of always updating the spinner when the imageView is updated, or vice versa, you will write a helper method to take care of it for you. Create this helper method in PhotoCollectionViewCell.swift:

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

18. 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. Implement these two methods in PhotoCollectionViewCell.swift to reset the cell back to the spinning state:

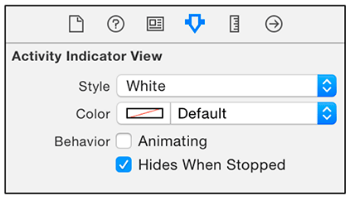
```
override func awakeFromNib() {
  super.awakeFromNib()

  update(with: nil)
}
override func prepareForReuse() {
  super.prepareForReuse()
  update(with: nil)
}
```

19. You will use a prototype cell to set up the interface for the collection view cell in the storyboard, just as you did for ItemCell. 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 Soo Greyhounds, you only need one kind of cell: the PhotoCollectionViewCell that displays a photo. Open Main.storyboard and select the collection view cell. In the identity inspector, change the Class to PhotoCollectionViewCelL, in the attributes inspector set Identifier to PhotoCollectionViewCell.

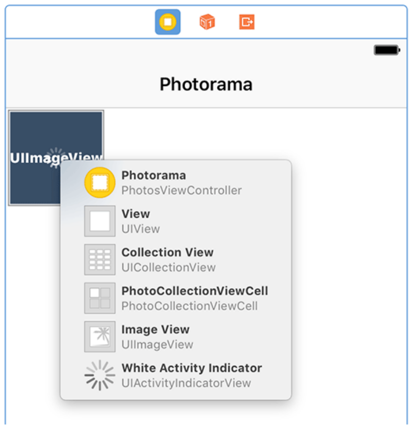

Custom Class
Class: PhotoCollectionViewCell
Module: Current – Photorama

20. 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.

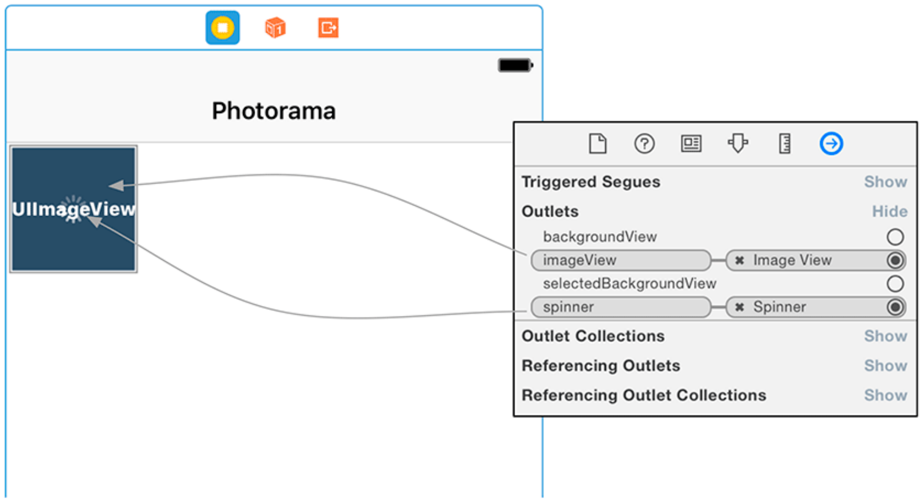
21. Next, drag an activity indicator view on top of the image view. Add constraints to center the activity indicator view both horizontally and vertically with the image view. Open its attributes inspector and select Hides When Stopped:


Activity Indicator View
Style: White
Color: Default
Behavior: ☐ Animating
☑ Hides When Stopped

22. 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:



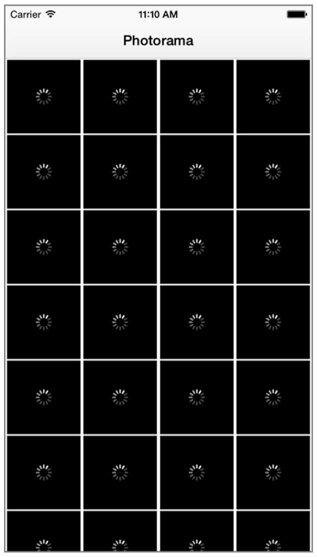
23. With the cell selected, open the connections inspector and connect the imageView and spinner properties to the image view and activity indicator view on the canvas:



24. Next, open PhotoDataSource.swift and update the data source method to use the PhotoCollectionViewCell:

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

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

  return cell
}
```

25. Build and run the application. When the latest photos request completes, you will see the activity indicator views all spinning:



26. In PhotosViewController.swift, have the class conform to the UICollectionViewDelegate protocol:

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

27. Update viewDidLoad() to set the PhotosViewController as the delegate of the collection view:

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

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

28. 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)
    }
  }
}
```

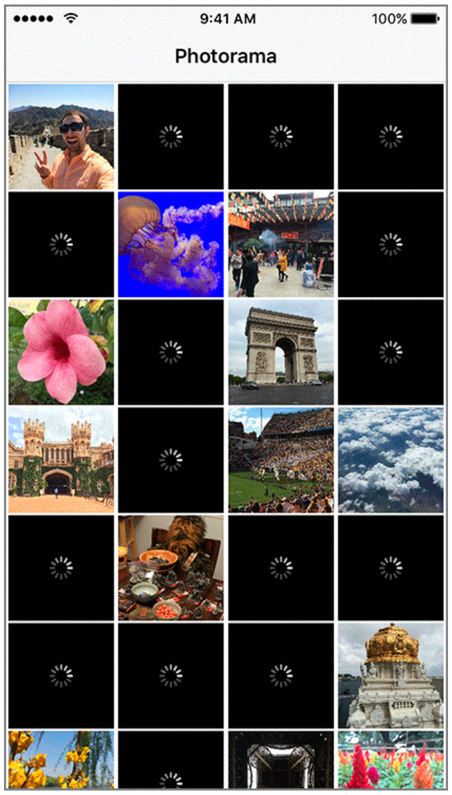
29. Let’s fix the error you saw when finding the index of photo in the photos array. The index(of:) method works by comparing the item that you are looking for to each of the items in the collection. It does this using the == operator. Types that conform to the Equatable protocol must implement this operator, and Photo does not yet conform to Equatable. 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
  }
}
```

30. Extensions provide a great mechanism for grouping related pieces of functionality. They can make the code more readable and help with long-term maintainability of your code base. One common chunk of functionality that is often grouped into an extension is conformance to a protocol along with the methods of that protocol. Update Photo.swift to use an extension to conform to the Equatable protocol:

```
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
    }
}
```

31. Build and run the application. The image data will download for the cells visible onscreen. Scroll down to make more cells visible. At first, you will see the activity indicator views spinning, but soon the image data for those cells will load.



32. If you scroll back up, you will see a delay in loading the image data for the previously visible cells. This is because whenever a cell comes onscreen, the image data is redownloaded. To fix this, you will implement image caching, similar to what you did in the Homepwner application.

33. Create a new Swift file named ImageStore. In ImageStore.swift, define the ImageStore class and add a property that is an instance of NSCache.

```
~~import Foundation~~
import UIKit

class ImageStore {

   let cache = NSCache<NSString,UIImage>()

}
```

34. Now implement three methods for adding, retrieving, and deleting an image from the cache dictionary.

```
class ImageStore {
  let cache = NSCache<NSString,UIImage>()
    func imageURL(forKey key: String) -> URL {

      let documentsDirectories =
        FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
      let documentDirectory = documentsDirectories.first!

      return documentDirectory.appendingPathComponent(key)
    }

  func setImage(_ image: UIImage, forKey key: String) {
    cache.setObject(image, forKey: key as NSString)

      // Create full URL for image
      let url = imageURL(forKey: key)

      // Turn image into JPEG data
      if let data = UIImageJPEGRepresentation(image, 0.5) {
        // Write it to full URL
        let _ = try? data.write(to: url, options: [.atomic])
      }
  }
  func image(forKey key: String) -> UIImage? {
      if let existingImage = cache.object(forKey: key as NSString) {
        return existingImage
      }
      let url = imageURL(forKey: key)
      guard let imageFromDisk = UIImage(contentsOfFile: url.path) else {
        return nil
      }
      cache.setObject(imageFromDisk, forKey: key as NSString)
      return imageFromDisk
  }
  func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)

      let url = imageURL(forKey: key)
      do {
        try FileManager.default.removeItem(at: url)
      } catch let deleteError {
        print("Error removing the image from disk: \(deleteError)")    }
  }
}
```

35. Back in Soo Greyhounds, open PhotoStore.swift and give it a property for an ImageStore.

```
class PhotoStore {
   let imageStore = ImageStore()
```

36. Then update fetchImage(for:completion:) to save the images using the imageStore.

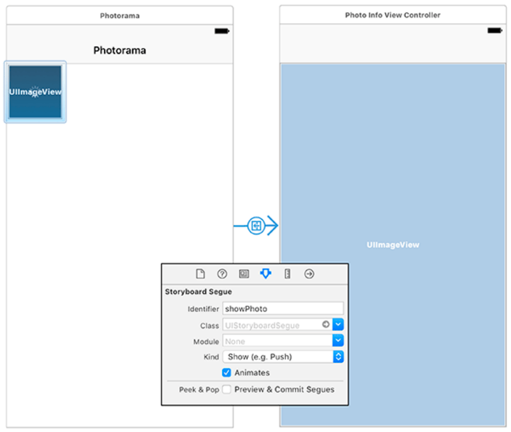
```
func fetchImage(for photo: Photo, completion: @escaping (ImageResult) -> Void) {
   let photoKey = photo.photoID
   if let image = imageStore.image(forKey: photoKey) {
      OperationQueue.main.addOperation {
         completion(.success(image))
      }
      return
   }

   let photoURL = photo.remoteURL
   let request = URLRequest(url: photoURL)

   let task = session.dataTask(with: request) {
      (data, response, error) -> Void in

      let result = self.processImageRequest(data: data, error: error)

      if case let .success(image) = result {
         self.imageStore.setImage(image, forKey: photoKey)
      }

      OperationQueue.main.addOperation {
         completion(result)
      }
   }
   task.resume()
}
```

37. Build and run the application. Now when the image data is downloaded, it will be saved to the filesystem. The next time that photo is requested, it will be loaded from the filesystem if it is not currently in memory.

38. Create a new Swift file named PhotoInfoViewController, declare the PhotoInfoViewController class, and add an imageView outlet.

```
~~import Foundation~~
import UIKit
class PhotoInfoViewController: UIViewController {
   @IBOutlet var imageView: UIImageView!
}
```

39. Now set up the interface for this view controller. Open Main.storyboard and drag a new View Controller onto the canvas from the object library. With this view controller selected, open its identity inspector and change the Class to PhotoInfoViewController. When the user taps on one of the collection view cells, the application will navigate to this new view controller. Control-drag from the PhotoCollectionViewCell to the Photo Info View Controller and select the Show segue. With the new segue selected, open its attributes inspector and give the segue an Identifier of showPhoto



Storyboard Segue
Identifier: showPhoto
Class: UIStoryboardSegue
Module: None
Kind: Show (e.g. Push)
☑ Animates
Peek & Pop ☐ Preview & Commit Segues

40. Add an image view to the Photo Info View Controller's view. Set up its Auto Layout constraints to pin the image view to all four sides. Open the attributes inspector for the image view and set its Content Mode to Aspect Fit.

41. Finally, connect the image view to the imageView outlet.

42. When the user taps a cell, the showPhoto segue will be triggered. At this point, the PhotosViewController will need to pass both the Photo and the PhotoStore to the PhotoInfoViewController. Open PhotoInfoViewController.swift and add two properties.

```
class PhotoInfoViewController: UIViewController {

    @IBOutlet var imageView: UIImageView!

    var photo: Photo! {
        didSet {
            navigationItem.title = photo.title
        }
    }
    var store: PhotoStore!
}
```

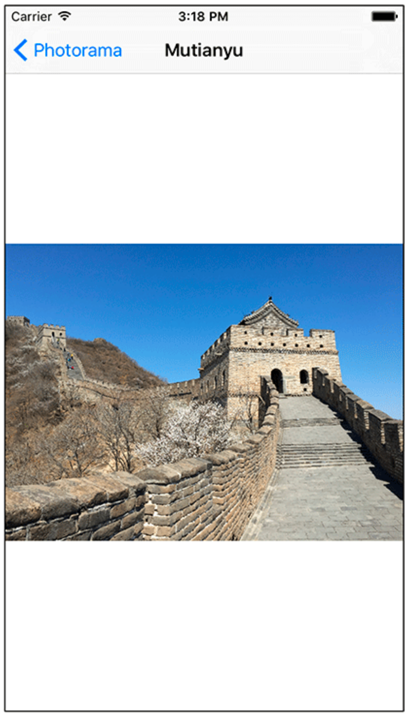
43. When photo is set on this view controller, the navigation item will be updated to display the name of the photo. Now override viewDidLoad() to set the image on the imageView when the view is loaded.

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

  store.fetchImage(for: photo) { (result) -> Void in
    switch result {
    case let .success(image):
      self.imageView.image = image
    case let .failure(error):
      print("Error fetching image for photo: \(error)")
    }
  }
}
```

44. In PhotosViewController.swift, implement prepare(for:sender:) to pass along the photo and the store.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  switch segue.identifier {
  case "showPhoto"?:
    if let selectedIndexPath =
        collectionView.indexPathsForSelectedItems?.first {

      let photo = photoDataSource.photos[selectedIndexPath.row]

      let destinationVC = segue.destination as! PhotoInfoViewController
      destinationVC.photo = photo
      destinationVC.store = store
    }
  default:
    preconditionFailure("Unexpected segue identifier.")
  }
}
```

45. Build and run the application. After the web service request has finished, tap on one of the photos to see it in the new view controller


Carrier
3:18 PM
Photorama
Mutianyu