Core Data Relationships-6

In this assignment you will incorporate Core Data relationships by adding the ability for users to tag photos with labels such as “Goal,” “Playoffs,” or “Forward.” Users will be able to add one or more tags to photos and also create their own custom tags.

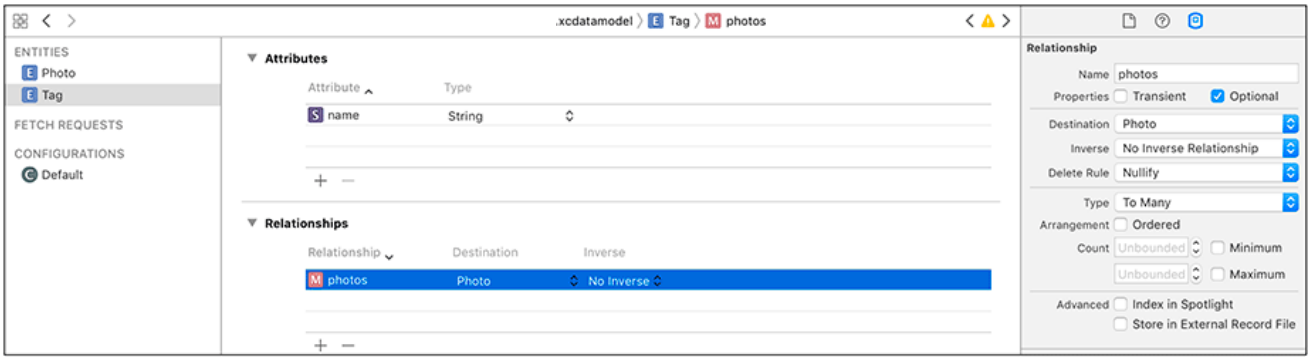
1. Create a new repository on GitHub and add your instructors as collaborators.

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. Open SooGreyhounds.xcdatamodeld and add another entity called Tag. Give it an attribute called name of type String. Tag will allow users to tag photos.

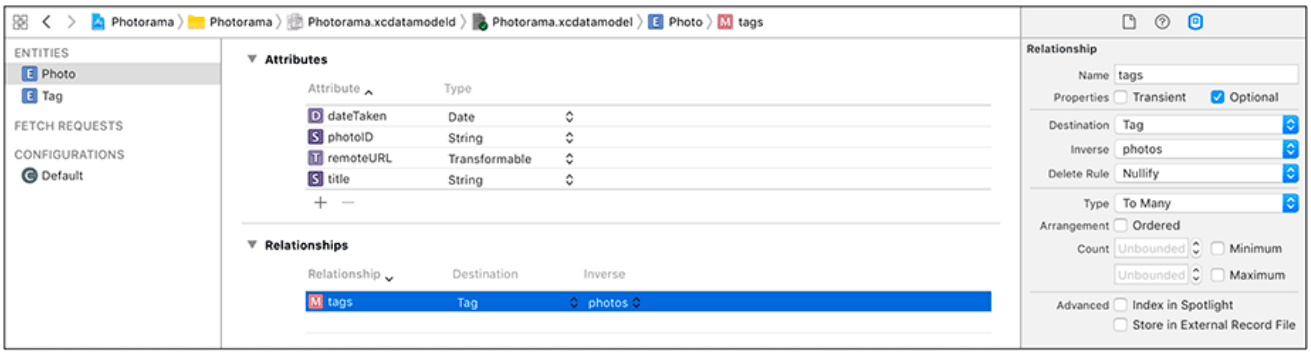
4. Unlike with the Photo entity you created previously, you will not generate an NSManagedObject subclass for the Tag entity. Instead, you will let Xcode autogenerate a subclass for you through a feature called code generation. If you do not need any custom behavior for your Core Data entity, letting Xcode generate your subclass for you is quite helpful. Select your Tag entity and open the data model inspector. Ensure “Class Definition” is selected for Codegen.

5. When these relationships are set up, you will be able to ask a Photo object for the set of Tag objects that it is associated with and ask a Tag object for the set of Photo objects that it is associated with. To add these two relationships to the model file, first select the Tag entity and click the + button in the Relationships section. Click in the Relationship column and enter photos. In the Destination column, select Photo. In the data model inspector, change the Type dropdown from To One to To Many.



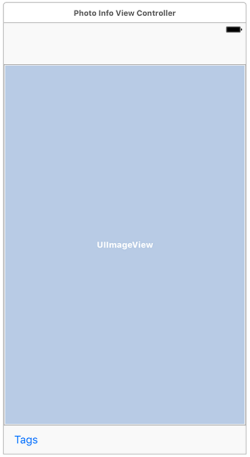
6. Next, select the Photo entity. Add a relationship named tags and pick Tag as its destination. In the data model inspector, change the Type dropdown to To Many and uncheck its Optional checkbox.

7. Now that you have two unidirectional relationships, you can make them into an inverse relationship. An inverse relationship is a bidirectional relationship between two entities. With an inverse relationship set up between Photo and Tag, Core Data can ensure that your object graph remains in a consistent state when any changes are made. To create the inverse relationship, click the dropdown next to Inverse in the data model inspector and change it from No Inverse Relationship to photos. (You can also make this change in the Relationships section in the editor area by clicking No Inverse in the Inverse column and selecting photos.)



8. Now that the model has changed for the Photo entity, you will need to regenerate the Photo+CoreDataProperties.swift file. From the project navigator, select and delete the Photo+CoreDataProperties.swift file. Make sure to select Move to Trash when prompted. Open SooGreyhounds.xcdatamodeld and select the Photo entity. From the Editor menu, select Create NSManagedObject Subclass…. On the next screen, check the box for SooGreyhounds and click Next. Check the box for the Photo entity and click Next. Make sure you are creating the file in the same directory as the Photo+CoreDataClass.swift file; this will ensure that Xcode will only create the necessary Photo+CoreDataProperties.swift file. Once you have confirmed this, click Create.

9. Open Main.storyboard and navigate to the interface for Photo Info View Controller. Add a toolbar to the bottom of the view. Update the Auto Layout constraints so that the toolbar is anchored to the bottom, just as it was in Homepwner. The bottom constraint for the imageView should be anchored to the top of the toolbar instead of the bottom of the superview. Add a UIBarButtonItem to the toolbar, if one is not already present, and give it a title of Tags. Your interface will look like.



10. Create a new Swift file named TagsViewController. Open this file and declare the TagsViewController class as a subclass of UITableViewController. Import UIKit and CoreData in this file.

```
~~import Foundation~~
import UIKit
import CoreData

class TagsViewController: UITableViewController {
}
```

11. Give the TagsViewController class a property to reference the PhotoStore as well as a specific Photo. You will also need a property to keep track of the currently selected tags, which you will track using an array of IndexPath instances.

```
class TagsViewController: UITableViewController {
   var store: PhotoStore!
   var photo: Photo!

   var selectedIndexPaths = [IndexPath]()
}
```

12. The data source for the table view will be a separate class. This class will be responsible for displaying the list of tags in the table view. Create a new Swift file named TagDataSource.swift. Declare the TagDataSource class and implement the table view data source methods.

```
~~import Foundation~~
import UIKit
import CoreData

class TagDataSource: NSObject, UITableViewDataSource {
   var tags: [Tag] = []

   func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
      return tags.count
   }

   func tableView(_ tableView: UITableView,
                  cellForRowAt indexPath: IndexPath) -> UITableViewCell {

      let cell = tableView.dequeueReusableCell(withIdentifier: "UITableViewCell",
                                               for: indexPath)
      let tag = tags[indexPath.row]
      cell.textLabel?.text = tag.name
      return cell
   }
}
```

13. Open PhotoStore.swift and define a new result type at the top for use when fetching tags.

```
enum PhotosResult {
    case success([Photo])
    case failure(Error)
}

enum TagsResult {
    case success([Tag])
    case failure(Error)
}
```

14. Now define a new method that fetches all the tags from the view context.

```
func fetchAllTags(completion: @escaping (TagsResult) -> Void) {
    let fetchRequest: NSFetchRequest<Tag> = Tag.fetchRequest()
    let sortByName = NSSortDescriptor(key: #keyPath(Tag.name), ascending: true)
    fetchRequest.sortDescriptors = [sortByName]

    let viewContext = persistentContainer.viewContext
    viewContext.perform {
        do {
            let allTags = try fetchRequest.execute()
            completion(.success(allTags))
        } catch {
            completion(.failure(error))
        }
    }
}
```

15. Open TagsViewController.swift and set the dataSource for the table view to be an instance of TagDataSource.

```
class TagsViewController: UITableViewController {
    var store: PhotoStore!
    var photo: Photo!

    var selectedIndexPaths = [IndexPath]()

    let tagDataSource = TagDataSource()

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView.dataSource = tagDataSource
    }
}
```

16. Now fetch the tags and associate them with the tags property on the data source.

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

   tableView.dataSource = tagDataSource

   updateTags()
}

func updateTags() {
   store.fetchAllTags {
      (tagsResult) in

      switch tagsResult {
      case let .success(tags):
         self.tagDataSource.tags = tags
      case let .failure(error):
         print("Error fetching tags: \(error).")
      }

      self.tableView.reloadSections(IndexSet(integer: 0),
                                    with: .automatic)
   }
}
```

17. The TagsViewController needs to manage the selection of tags and update the Photo instance when the user selects or deselects a tag. In TagsViewController.swift, add the appropriate index paths to the selectedIndexPaths array.

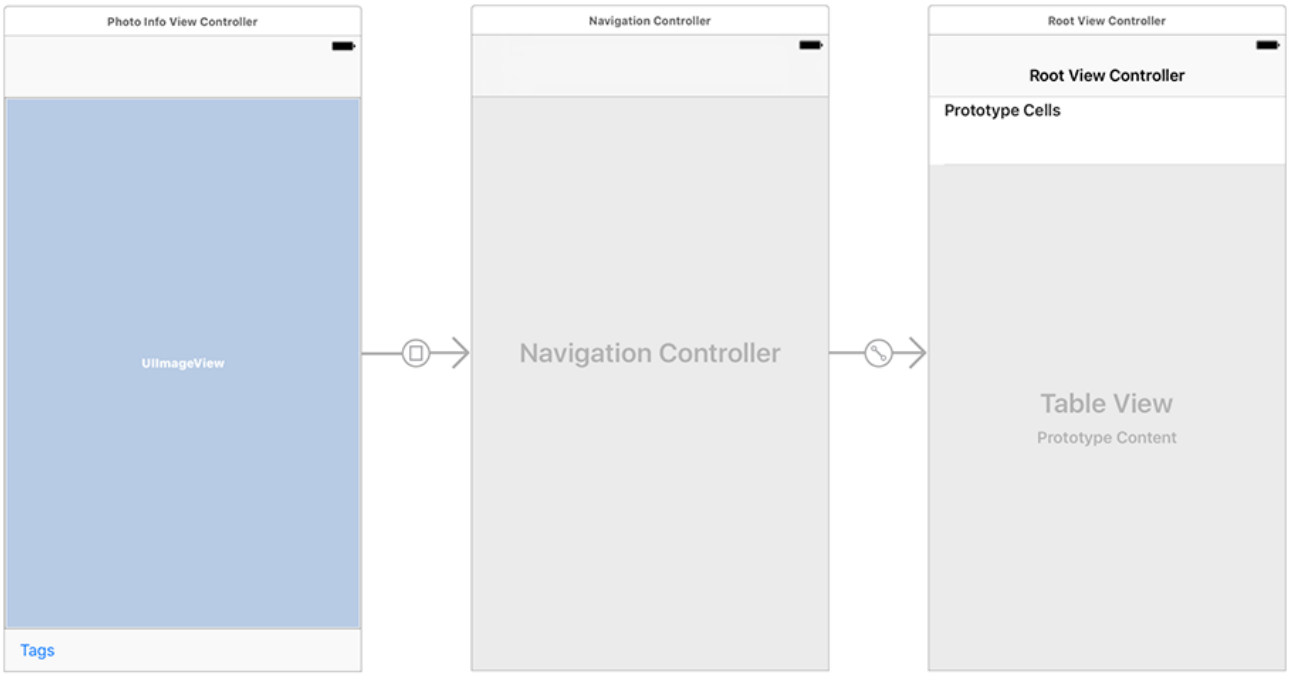
```
override func viewDidLoad() {
    super.viewDidLoad()

    tableView.dataSource = tagDataSource
    tableView.delegate = self

    updateTags()
}

func updateTags() {
    store.fetchAllTags {
        (tagsResult) in

        switch tagsResult {
        case let .success(tags):
            self.tagDataSource.tags = tags

            guard let photoTags = self.photo.tags as? Set<Tag> else {
                return
            }

            for tag in photoTags {
                if let index = self.tagDataSource.tags.index(of: tag) {
                    let indexPath = IndexPath(row: index, section: 0)
                    self.selectedIndexPaths.append(indexPath)
                }
            }
        case let .failure(error):
            print("Error fetching tags: \(error).")
        }

        self.tableView.reloadSections(IndexSet(integer: 0),
                                      with: .automatic)
    }
}
```

18. Now add the appropriate UITableViewDelegate methods to handle selecting and displaying the checkmarks.

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {

  let tag = tagDataSource.tags[indexPath.row]

  if let index = selectedIndexPaths.index(of: indexPath) {
    selectedIndexPaths.remove(at: index)
    photo.removeFromTags(tag)
  } else {
    selectedIndexPaths.append(indexPath)
    photo.addToTags(tag)
  }

  do {
    try store.persistentContainer.viewContext.save()
  } catch {
    print("Core Data save failed: \(error).")
  }

  tableView.reloadRows(at: [indexPath], with: .automatic)
}

override func tableView(_ tableView: UITableView,
                        willDisplay cell: UITableViewCell,
                        forRowAt indexPath: IndexPath) {

  if selectedIndexPaths.index(of: indexPath) != nil {
    cell.accessoryType = .checkmark
  } else {
    cell.accessoryType = .none
  }
}
```

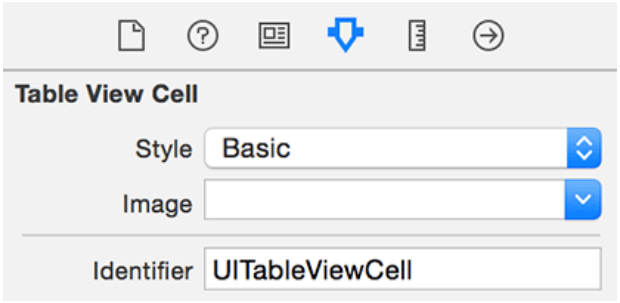
19. Let's set up TagsViewController to be presented modally when the user taps the Tags bar button item on the PhotoInfoViewController. Open Main.storyboard and drag a Navigation Controller onto the canvas. This should give you a UINavigationController with a root view controller that is a UITableViewController. If the root view controller is not a UITableViewController, delete the root view controller, drag a Table View Controller onto the canvas, and make it the root view controller of the Navigation Controller.

20. Control-drag from the Tags item on Photo Info View Controller to the new Navigation Controller and select the Present Modally segue type. Open the attributes inspector for the segue and give it an Identifier named showTags.

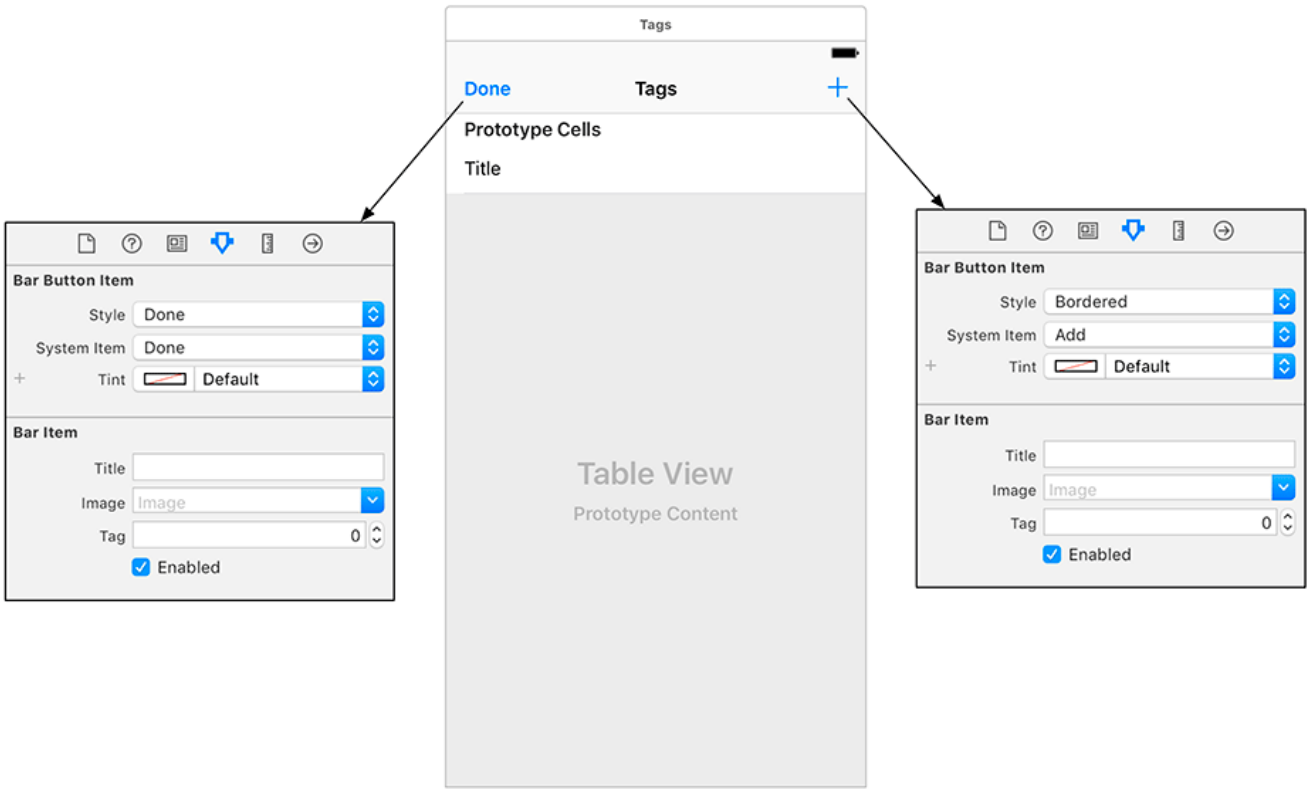


21. Select the Root View Controller that you just added to the canvas and open its identity inspector. Change its Class to TagsViewController. This new view controller does not have a navigation item associated with it, so find Navigation Item in the object library and drag it onto the view controller. Select the new navigation item and change it's title to Tags.

22. The UITableViewCell on the Tags View Controller interface needs to match what the TagDataSource expects. It needs to use the correct style and have the correct reuse identifier. Select the UITableViewCell. (It might be easier to select in the document outline.) Open its attributes inspector. Change the Style to Basic and set the Identifier to UITableViewCell.



23. Now, the Tags View Controller needs two bar button items on its navigation bar: a Done button that dismisses the view controller and a + button that allows the user to add a new tag. Drag a bar button item to the left and right bar button item slots for the Tags View Controller. Set the left item to use the Done style and system item. Set the right item to use the Bordered style and Add system item.



24. Create and connect an action for each of these items to the TagsViewController. The Done item should be connected to a method named done(_:), and the + item should be connected to a method named addNewTag(_:). The two methods in TagsViewController.swift will be:

```
@IBAction func done(_ sender: UIBarButtonItem) {

}

@IBAction func addNewTag(_ sender: UIBarButtonItem) {

}
```

25. The implementation of done(_:) is simple: The view controller just needs to be dismissed. Implement this functionality in done(_:).

```
@IBAction func done(_ sender: UIBarButtonItem) {
    presentingViewController?.dismiss(animated: true,
                                      completion: nil)
}
```

26. When the user taps the + item, an alert will be presented that will allow the user to type in the name for a new tag. Set up and present an instance of UIAlertController in addNewTag(_:).

```
@IBAction func addNewTag(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: "Add Tag",
                                            message: nil,
                                            preferredStyle: .alert)

    alertController.addTextField {
        (textField) -> Void in
        textField.placeholder = "tag name"
        textField.autocapitalizationType = .words
    }

    let okAction = UIAlertAction(title: "OK", style: .default) {
        (action) -> Void in

    }
    alertController.addAction(okAction)

    let cancelAction = UIAlertAction(title: "Cancel",
                                     style: .cancel,
                                     handler: nil)
    alertController.addAction(cancelAction)

    present(alertController,
            animated: true,
            completion: nil)
}
```

27. Update the completion handler for the okAction to insert a new Tag into the context. Then save the context, update the list of tags, and reload the table view section.

```
let okAction = UIAlertAction(title: "OK", style: .default) {
   (action) -> Void in

   if let tagName = alertController.textFields?.first?.text {
      let context = self.store.persistentContainer.viewContext
      let newTag = NSEntityDescription.insertNewObject(forEntityName: "Tag",
                                          into: context)
      newTag.setValue(tagName, forKey: "name")

      do {
         try self.store.persistentContainer.viewContext.save()
      } catch let error {
         print("Core Data save failed: \(error)")
      }
      self.updateTags()
   }
}
alertController.addAction(okAction)
```

28. When the Tags bar button item on PhotoInfoViewController is tapped, the PhotoInfoViewController needs to pass along its store and photo to the TagsViewController. Open PhotoInfoViewController.swift and implement prepare(for:).

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
   switch segue.identifier {
   case "showTags"?:
      let navController = segue.destination as! UINavigationController
      let tagController = navController.topViewController as! TagsViewController

      tagController.store = store
      tagController.photo = photo
   default:
      preconditionFailure("Unexpected segue identifier.")
   }
}
```

29. Build and run the application. Navigate to a photo and tap the Tags item on the toolbar at the bottom. The TagsViewController will be presented modally. Tap the + item, enter a new tag, and select the new tag to associate it with the photo.

30. You are going to update processPhotosRequest(data:error:) to use a background task. Background tasks are an asynchronous operation, so you will need to update the method signature to use a completion handler. Open PhotoStore.swift and update processPhotosRequest(data:error:) to take in a completion handler. You will have some errors in the code due to the signature change; you will fix these shortly.

```
~~private func processPhotosRequest(data: Data?, error: Error?) -> PhotosResult {~~
private func processPhotosRequest(data: Data?,
                                  error: Error?,
                                  completion: @escaping (PhotosResult) -> Void) {
    guard let jsonData = data else {
        return .failure(error!)
    }

    return SooGreyhoundsAPI.photos(fromJSON: jsonData,
                                   into: persistentContainer.viewContext)
}
```

31. If there is no data, you will need to call the completion handler, passing in the failure error instead of directly returning. Update the guard statement to pass along the failure.

```
private func processPhotosRequest(data: Data?,
                                  error: Error?,
                                  completion: @escaping (PhotosResult) -> Void) {
    guard let jsonData = data else {
~~        return .failure(error!)~~
        completion(.failure(error!))
        return
    }

    return SooGreyhoundsAPI.photos(fromJSON: jsonData,
                                   into: persistentContainer.viewContext)
}
```

32. Now you can add in the code for the background task. NSPersistentContainer has a method to perform a background task. This method takes in a closure to call, and this closure vends a new NSManagedObjectContext to use. Update processPhotosRequest(data:error:completion:) to kick off a new background task.

```
private func processPhotosRequest(data: Data?,
                                  error: Error?,
                                  completion: @escaping (PhotosResult) -> Void) {
    guard let jsonData = data else {
        completion(.failure(error!))
        return
    }

    return SooGreyhoundsAPI.photos(fromJSON: jsonData,
                                   into: persistentContainer.viewContext)

    persistentContainer.performBackgroundTask {
        (context) in

    }
}
```

33. Within the background task, you will do essentially the same thing you did before. The SooGreyhoundsAPI struct will ingest the JSON data and convert it to Photo instances. Then you will save the context so that the insertions persist. Update the background task in processPhotosRequest(data:error:completion:) to do this.

```
persistentContainer.performBackgroundTask {
    (context) in

    let result = SooGreyhoundsAPI.photos(fromJSON: jsonData, into: context)

    do {
        try context.save()
    } catch {
        print("Error saving to Core Data: \(error).")
        completion(.failure(error))
        return
    }
}
```

34. Update processPhotosRequest(data:error:completion:) to get the Photo instances associated with the viewContext and pass them back to the caller via the completion handler.

```
persistentContainer.performBackgroundTask {
   (context) in

   let result = SooGreyhoundsAPI.photos(fromJSON: jsonData, into: context)

   do {
      try context.save()
   } catch {
      print("Error saving to Core Data: \(error).")
      completion(.failure(error))
      return
   }

   switch result {
   case let .success(photos):
      let photoIDs = photos.map { return $0.objectID }
      let viewContext = self.persistentContainer.viewContext
      let viewContextPhotos =
         photoIDs.map { return viewContext.object(with: $0) } as! [Photo]
      completion(.success(viewContextPhotos))
   case .failure:
      completion(result)
   }
}
```

35. The final change you need to make is to update fetchLatestPhotos(completion:) to use the updated processPhotosRequest(data:error:completion:) method.

```
func fetchLatestPhotos(completion: @escaping (PhotosResult) -> Void) {

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

        var result = self.processPhotosRequest(data: data, error: error)

        if case .success = result {
            do {
                try self.persistentContainer.viewContext.save()
            } catch let error {
                result = .failure(error)
            }
        }

        OperationQueue.main.addOperation {
            completion(result)
        }

        self.processPhotosRequest(data: data, error: error) {
            (result) in

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

36. Build and run the application. Although the behavior has not changed, the application is no longer in danger of becoming unresponsive while new photos are being added. As the scale of your applications increases, handling Core Data entities somewhere other than the main queue as you have done here can result in huge performance wins.