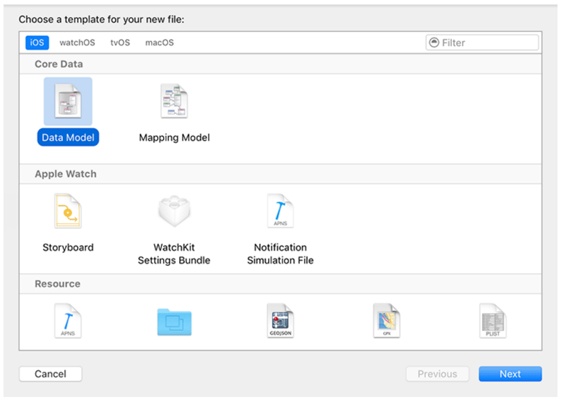
Core Data-5

In this assignment you will incorporate Core Data to manage the saving of your app's information and also add tags to the 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.

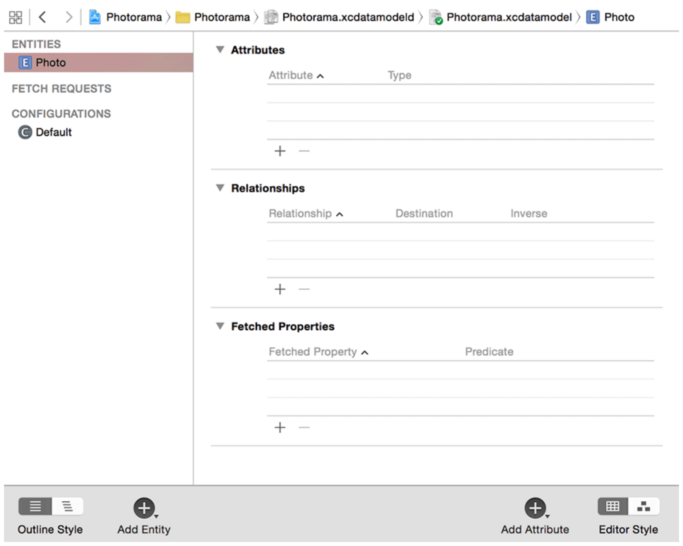
2. Open your Soo Greyhounds project, then open the source control navigator. Remove your previous labs remote repository and add your new repository as a remote.

3. Create a new file, but do not make it a Swift file like the ones you have created before. Instead, select iOS at the top and scroll down to the Core Data section. Select Data Model and give it the name "SooGreyhounds".



4. This will create the SooGreyhounds.xcdatamodeld file and add it to your project. Select this file from the project navigator and the editor area will reveal the UI for manipulating a Core Data model file.

5. Find the Add Entity button at the bottom left of the window and click it. A new entity will appear in the list of entities in the lefthand table. Double-click this entity and change its name to Photo.



6. Now your Photo entity needs attributes. Remember that these will be the properties of the Photo class. The necessary attributes are listed below. For each attribute, click the + button in the Attributes section and edit the Attribute and Type values.
   a) photoID is a String
   b) title is a String
   c) dateTaken is a Date
   d) remoteURL is a Transformable

7. With SooGreyhounds.xcdatamodeld still open, select the remoteURL attribute and open its Data Model inspector on the righthand side. Under the Attribute section, enter NSURL as the Custom Class. This will allow Core Data to do the transformation for you.

8. In the project navigator, select the Photo.swift file and delete it. When prompted, move it to the trash to make sure it does not still exist in the project directory.

9. Open SooGreyhounds.xcdatamodeld. Select the Photo entity and open the Data Model inspector. Locate the Codegen option and select Manual/None.

10. With the Photo entity still selected, open the Editor menu and 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 again. Finally, click Create. There will be a few errors in the project. You will fix those shortly.

11. The template will create two files for you: Photo+CoreDataClass.swift and Photo+CoreDataProperties.swift. The template places all of the attributes that you defined in the model file into Photo+CoreDataProperties.swift. If you ever change your entity in the model file, you can simply delete Photo+CoreDataProperties.swift and regenerate the NSManagedObject subclass. Xcode will recognize that you already have Photo+CoreDataClass.swift and will only re-create Photo+CoreDataProperties.swift. Open Photo+CoreDataProperties.swift and take a look at what the template created for you.

12. Open PhotoStore.swift and find fetchImage(for:completion:). This method expects the photoID and the remoteURL to be non-optional; however, Core Data models its attributes as optionals. Additionally, the URLRequest initializer expects a URL instance as its argument instead of an NSURL instance. Update the method to address these issues.

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

    guard let photoURL = photo.remoteURL else {
      preconditionFailure("Photo expected to have a remote URL.")
    }
    let request = URLRequest(url: photoURL as URL)

    …
}
```

13. To address the first issue, you are using a guard statement to unwrap the optional NSURL. To address the second issue, you bridge the NSURL instance to a URL instance using an as cast. The compiler knows that NSURL and URL are related, so it handles the bridging conversion. You have created your model graph and defined your Photo entity. The next step is to set up the persistent container, which will manage the interactions between the application and Core Data. There are still some errors in the project; you will fix them after you have added a Core Data persistent container instance.

14. Open PhotoStore.swift and import Core Data at the top of the file.

```
import UIKit
import CoreData
```

15. Also in PhotoStore.swift, add a property to hold on to an instance of NSPersistentContainer.

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

    let persistentContainer: NSPersistentContainer = {
        let container = NSPersistentContainer(name: "SooGreyhounds")
        container.loadPersistentStores { (description, error) in
            if let error = error {
                print("Error setting up Core Data (\(error)).")
            }
        }
        return container
    }()

    …
}
```

16. When an entity is created, it should be inserted into a managed object context. Open SooGreyhoundsAPI.swift and import CoreData.

```
import Foundation
import CoreData
```

17. Update the photo(fromJSON:) method to take in an additional argument of type NSManagedObjectContext and use this context to insert new Photo instances.

```
private static func photo(fromJSON json: [String : Any],
                    into context: NSManagedObjectContext) -> Photo? {
    guard
        let photoID = json["id"] as? String,
        let title = json["title"] as? String,
        let dateString = json["datetaken"] as? String,
        let photoURLString = json["url_h"] as? String,
        let url = URL(string: photoURLString),
        let dateTaken = dateFormatter.date(from: dateString) else {

            // Don't have enough information to construct a Photo
            return nil
    }

~~    return Photo(title: title, photoID: photoID, remoteURL: url, dateTaken: dateTaken)~~

    var photo: Photo!
    context.performAndWait {
        photo = Photo(context: context)
        photo.title = title
        photo.photoID = photoID
        photo.remoteURL = url as NSURL
        photo.dateTaken = dateTaken as NSDate
    }

    return photo
}
```

18. The photo(fromJSON:into:) method is called from the method photos(fromJSON:). Update this method to take in a context and pass it to the photo(fromJSON:into:) method.

```
static func photos(fromJSON data: Data,
            into context: NSManagedObjectContext) -> PhotosResult {
    do {
        ...
        var finalPhotos = [Photo]()
        for photoJSON in photosArray {
            if let photo = photo(fromJSON: photoJSON, into: context) {
                finalPhotos.append(photo)
            }
        }
    ...
}
```

19. You now need to pass the viewContext to the SooGreyhoundsAPI struct once the web service request successfully completes. Open PhotoStore.swift and update processPhotosRequest(data:error:).

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

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

20. Build and run the application now that all errors have been addressed. Although the behavior remains unchanged, the application is now backed by Core Data.

21. Recall that NSManagedObject changes do not persist until you tell the context to save these changes. Open PhotoStore.swift and update fetchLatestPhotos(completion:) to save the changes to the context after Photo entities have been inserted into the context.

```
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

      ~~let~~ 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)
      }
   }
   task.resume()
}
```

22. You want to sort the returned instances of Photo by dateTaken in descending order. To do this, you will instantiate an NSFetchRequest for requesting “Photo” entities. Then you will give the fetch request an array of NSSortDescriptor instances. For SooGreyhounds, this array will contain a single sort descriptor that sorts photos by their dateTaken properties. Finally, you will ask the managed object context to execute this fetch request. In PhotoStore.swift, implement a method that will fetch the Photo instances from the view context.

```
func fetchAllPhotos(completion: @escaping (PhotosResult) -> Void) {
  let fetchRequest: NSFetchRequest<Photo> = Photo.fetchRequest()
  let sortByDateTaken = NSSortDescriptor(key: #keyPath(Photo.dateTaken),
                                         ascending: true)
  fetchRequest.sortDescriptors = [sortByDateTaken]

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

23. Open PhotosViewController.swift and add a new method that will update the data source with all of the photos.

```
private func updateDataSource() {
  store.fetchAllPhotos {
    (photosResult) in

    switch photosResult {
    case let .success(photos):
      self.photoDataSource.photos = photos
    case .failure:
      self.photoDataSource.photos.removeAll()
    }
    self.collectionView.reloadSections(IndexSet(integer: 0))
  }
}
```

24. Now update viewDidLoad() to call this method to fetch and display all of the photos saved to Core Data.

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

   collectionView.dataSource = photoDataSource
   collectionView.delegate = self

   store.fetchLatestPhotos {
      (photosResult) -> Void in

~~      switch photosResult {~~
~~      case let .success(photos):~~
~~         print("Successfully found \(photos.count) photos.")~~
~~         self.photoDataSource.photos = photos~~
~~      case let .failure(error):~~
~~         print("Error fetching Latest photos: \(error)")~~
~~         self.photoDataSource.photos.removeAll()~~
~~      }~~
~~      self.collectionView.reloadSections(IndexSet(integer: 0))~~

      self.updateDataSource()
   }
}
```

25. To prevent creating duplicate photos when running the app multiple times we will use the photos unique id to compare what we have in our local database. You need a way to tell the fetch request that it should not return all photos but instead only the photos that match some specific criteria. In this case, the specific criteria is “only photos that have this specific identifier,” of which there should either be zero or one photo. In Core Data, this is done with a predicate. In SooGreyhoundsAPI.swift, update photo(fromJSON:into:) to check whether there is an existing photo with a given ID before inserting a new one.

```
private static func photo(fromJSON json: [String : Any],
                          into context: NSManagedObjectContext) -> Photo? {
  guard
    let photoID = json["id"] as? String,
    let title = json["title"] as? String,
    let dateString = json["datetaken"] as? String,
    let photoURLString = json["url_h"] as? String,
    let url = URL(string: photoURLString),
    let dateTaken = dateFormatter.date(from: dateString) else {

      // Don't have enough information to construct a Photo
      return nil
  }

  let fetchRequest: NSFetchRequest<Photo> = Photo.fetchRequest()
  let predicate = NSPredicate(format: "\(#keyPath(Photo.photoID)) == \(photoID)")
  fetchRequest.predicate = predicate

  var fetchedPhotos: [Photo]?
  context.performAndWait {
    fetchedPhotos = try? fetchRequest.execute()
  }
  if let existingPhoto = fetchedPhotos?.first {
    return existingPhoto
  }

  var photo: Photo!
  context.performAndWait {
    photo = Photo(context: context)
    photo.title = title
    photo.photoID = photoID
    photo.remoteURL = url as NSURL
    photo.dateTaken = dateTaken as NSDate
  }

  return photo

}
```

26. Duplicate photos will no longer be inserted into Core Data. Build and run the application. The photos will appear just as they did before introducing Core Data. Close the application using the Home button (or Shift-Command-H in the simulator). Launch the application again and you will see the photos that Core Data saved in the collection view.

27. There is one last small problem to address: The user will not see any photos appear in the collection view unless the web service request completes. If the user has slow network access, it might take up to 60 seconds (which is the default timeout interval for the request) to see any photos. It would be best to see the previously saved photos immediately on launch and then refresh the collection view once new photos are fetched from SooGreyhounds. Go ahead and do this. In PhotosViewController.swift, update the data source as soon as the view is loaded.

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

   collectionView.dataSource = photoDataSource
   collectionView.delegate = self

   updateDataSource()

   store.fetchLatestPhotos {
      (photosResult) -> Void in

      self.updateDataSource()
   }
}
```

28. Run your app and ensure it is working properly.