Assignment VI:

Core Data SPoT

Objective

In this series of assignments, you will create an application that lets users tour Stanford through photos. The first assignment was to create a navigation-based application to let users browse different categories of spots at Stanford, then click on any they are interested in to see a photo of it. The second assignment used GCD and file system cacheing to improve the performce.

This assignment will focus on retrieving data out of a Core Data database (rather than fetching it from the network directly).

Even though this application is substantially similar to last week's it is recommended that you start fresh with a completely new application because your underlying data structure will be completely different (Core Data rather than arrays of dictionaries). You may still want to drag some of your classes from last week in on occasion (especially your view controller that shows a photo in a scrolling view).

All the data you need will be downloaded from Flickr.com using Flickr's API.

This assignment is due by the start of lecture next Thursday (not *this* Thursday, *next* Thursday).

Be sure to check out the **Hints** section below!

Materials

- You will need the `FlickrFetcher` helper code (download from the same place you found this document).
- You will need to obtain a Flickr API key. A free Flickr account is just fine (you won't be posting photos, just querying them).

Required Tasks

1. Create a Core Data-managed object database model in Xcode suitable for storing any information you queried from Flickr in the last assignment that you feel you will need to perform the rest of the Required Tasks in this assignment. Take the time to give some thought to what Entities, Attributes and Relationships your database will need before you dive in.
2. Recreate the user-interface from last week's assignment but rely entirely on the above-mentioned Core Data database to present it.
3. Add a thumbnail image (`FlickrFetcherPhotoFormatSquare`) of each spot to every row in any table view that shows a list of spots. In addition ...
   a. Don't ask Flickr for the image data of thumbnails that never appear on screen (i.e. fetch thumbnail image data only on demand).
   b. Don't block your main thread waiting for Flickr to give you a thumbnail's data.
   c. Never ask Flickr for the same thumbnail twice (cache the thumbnails' data in your Core Data database).
4. The Recents tab should continue to meet all the requirements of the previous assignment. All Recents information should be stored in Core Data (i.e. do not use `NSUserDefaults` for Recents anymore).
5. This week's version should still never block the main thread with any Flickr fetches and should continue to cache photo (non-thumbnail) image data in the filesystem in the same way as last week's assignment.
6. When the user refreshes the table view (with the `UIRefreshControl`), you should refetch the data from Flickr in a thread and update your Core Data database and then update the UI to show any new data. This refresh should automatically invoke upon launch of your application if the Core Data database is empty.
7. Your application must continue to work properly in portrait and landscape and on iPhone 4, iPhone 5 and iPad. It must also verify that it continues to work properly on a physical device (of your choice).

Hints

These hints are not required tasks. They are completely optional. Following them may make the assignment a little easier (no guarantees though!).

1. You can get/set your Entities' Attributes using `valueForKey:/setValueForKey:` if you prefer, but, especially when it comes to loading up the database, you're probably going to want to create a custom subclass of `NSManagedObject` and add an Objective-C category for your own Entity-specific code.
2. Do not use an attribute in your data model named `description`. This will be tempting, but it will cause problems (because of `NSObject`'s `description` method).
3. Do not use an attribute in your data model named `id`. You can imagine the problems this might cause in Objective-C.
4. Remember that in order to fetch a list of a certain type of object (like a "tag" or a "photo"), there must be a representation for that object (an Entity) in the managed object model you create in Xcode's data modeler. Or, in database terms, there must be a "table" for it. For example, you cannot fetch a list of "photos" into a table view if there is no Entity that represents a "photo" in the database.
5. If you change your object model (and you will likely do that numerous times as you iterate on your schema), be sure to delete your application from your device or simulator before running it with a new schema so that your old database (with the old schema) gets deleted. You do this by pressing and holding on the application icon on the home screen of the device or simulator until it jiggles, then pressing the `X` that appears in the corner of the application's icon. If you fail to do this when you change your schema, your program will crash (with complaints in the console about incompatible database descriptions).
6. When you are writing the code to transfer data from a Flickr fetch to your Core Data database, remember to query your object model first to see if an object already exists before creating a new one or you will get lots of duplicate objects. You query objects using `executeFetchRequest:error:` on an `NSManagedObjectContext` instance with an `NSFetchRequest` containing an appropriate `NSEntityDescription` and `NSPredicate`. You create new objects using `NSEntityDescription`'s class method `insertNewObjectForEntityForName:inManagedObjectContext:`.
7. Note that if you have a two-way relationship in your model, setting one side automatically sets the other side properly for you.
8. "To many" relationships are represented simply as an `NSSet` of `NSManagedObject` instances. So to set the value of a property that is a to-many relationship from the "many" side (if you need to do this), just create an `NSMutableSet`, add `NSManagedObject` instances to it, then set the property to that `NSMutableSet`.

9. You do not have to use the provided `CoreDataTableViewController` for this assignment, but it's extremely likely you'll want to! Its implementation is mostly just copy/pasted from the documentation of `NSFetchedResultsController`. It's very easy to subclass and use. The most important thing you must do is set its `fetchedResultsController` property, then use it in your implementation of the UITableViewDataSource protocol whenever you need to get the managed object corresponding to a given index path.
10. If you find yourself writing a lot of code to make your table views work, then you are probably headed down the wrong path. Let `NSFetchedResultsController` and `CoreDataTableViewController` do the work for you.
11. `NSFetchedResultsController` has a property called `fetchedObjects` which is the array of `NSManagedObject`s it fetched. Checking whether this array is empty (empty, not `nil`) is probably a good way to figure out whether to "auto-fetch" when your application runs and there is no data in the database. If `fetchedObjects` is `nil`, it means the fetch has not been attempted yet. Pick a spot in your View Controller's Lifecycle where the fetch has happened before you check to see if the fetch came back with no results. When you implement methods in your View Controller's Lifecycle, be sure not to forget to invoke your `super`'s implementation so that `CoreDataTableViewController` can do what it wants to do too.
12. The sorting in `CoreDataTableViewController`-based view controllers should happen automatically as part of the `NSFetchRequest` you use to create your `NSFetchedResultsController`. Just make sure it has the right `NSSortDescriptor`.
13. Both the `predicate` and the `sortDescriptors` for your Recents `CoreDataTableViewController`'s `NSFetchedResultsController`'s `NSFetchRequest` are going to depend on knowing when a photo in the database was last viewed, so it might be a good idea to have an attribute for that in your schema and make sure it gets set properly.
14. Note that your user-interface will automatically update when you make modifications to the database that it is hooked up to. That is the wonder of `NSFetchedResultsController` and Core Data's use of the key-value observing mechanism.
15. It is probably a good idea to invoke all Core Data operations in the main thread for this assignment. It may be more "correct" to do some of those operations in other threads, but that requires a much better understanding of how Core Data uses threads than we are able to cover in this course (though you are more than welcome to try to figure this out). `NSManagedObjectContext`'s `performBlock:` would be the best way to get Core Data operations onto the right thread.
16. The `FlickrFetcher` method `urlForPhoto:format:` can be used to get a url for the photo's image or for its thumbnail. The only difference is the format (`Square` vs

`Large` or `Original`). You're probably going to want to store both of these urls into your database.

Evaluation

In all of the assignments this quarter, writing quality code that builds without warnings or errors, and then testing the resulting application and iterating until it functions properly is the goal.

Here are the most common reasons assignments are marked down:

- Project does not build.
- Project does not build without warnings.
- One or more items in the **Required Tasks** section was not satisfied.
- A fundamental concept was not understood.
- Code is sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible) for someone reading the code to understand due to lack of comments, poor variable/method names, poor solution structure, etc.
- UI is a mess. Things should be lined up and appropriately spaced to "look nice." Xcode gives you those dashed blue guidelines so there should be no excuse for things not being lined up, etc. Get in the habit of building aesthetically balanced UIs from the start of this course.
- Assignment was turned in late (you get 3 late days per quarter, so use them wisely).
- Incorrect or poor use of object-oriented design principles. For example, code should not be duplicated if it can be reused via inheritance or other object-oriented design methodologies.

Often students ask "how much commenting of my code do I need to do?" The answer is that your code must be easily and completely understandable by anyone reading it. You can assume that the reader knows the SDK, but should not assume that they already know the (or a) solution to the problem.

Extra Credit

If you do any Extra Credit items, don't forget to note what you did in your submission `README`.

1. Divide all tables of photos (except maybe the "All" table below) into alphabetical sections. This turns out to be easy in table views driven by Core Data, but only if you have an attribute in the database which is "that photo's section heading" and which sorts in exactly the same order as the photos themselves sort.
2. Allow users to delete photos (i.e. they no longer appear in tables). You should do it in a way that makes it so that the `UIRefreshControl` does not bring them back!
3. Have a row in your main table view called "All" which shows all the photos (not just the photos with a certain tag). Then have the table that appears when you click on "All" be divided into sections by tag (i.e., each section is one of the tags). The simplest way to add the "All" row is to update the information in your database a little bit rather than trying to special-case your table view code (though you will need a special case to turn on sections for that particular table of all photos since you don't want other tables of photos to have sections). Luckily "All" is alphabetically at the top of the list of types of places, but can you think of a way to modify your schema/fetch criteria to ensure that it is always the top item in the list even if it's not called "All" or if there were an "Abattoir" tag in the list?
4. Make one or more of your table views searchable. Check out `UISearchDisplayController` and `UISearchBar`.