Assignment VI:

Top Regions

Objective

In this assignment, we will enhance our Top Places application to store the information it fetches from Flickr into Core Data and then use that to our advantage to make some smart queries.

Be sure to review the Hints section below!

Materials

- You will need to have completed Assignment 5 to understand what's going on in this assignment. Your storyboards may well be very similar, but the internal implementation of all of your `UITableViewController`s will be different in this assignment.
- The class CoreDataTableViewController is essential to this assignment. You should definitely use it and, though it is not required, you should probably look at its implementation and try to understand what it is doing (look at the documentation for `NSFetchedResultsController` as you do so).
- FlickrFetcher is also still needed for this assignment.
- You will still need your Flickr API key. A free Flickr account is just fine (we're still just querying, not posting).

Required Tasks

1. Your application must work identically to last week except that where you are displaying the "top places" (as reported by Flickr), you are going to display the "top regions" in which photos have been taken. You will *calculate* the most popular regions from the data you gather periodically from the `URLforRecentGeoreferencedPhotos`.
2. The popularity of a region is determined by how many *different photographers* have taken a photo in that region among the photos you've downloaded from Flickr. Only show the 50 most popular regions in your UI (it is okay if the table temporarily shows more than 50 as data is loaded into the database, but re-set it to 50 occasionally).
3. The list of top regions must be sorted first by popularity (most popular first, of course) and secondarily by the name of the region. Display the number of different photographers who have taken a photo in that region as a subtitle in each row.
4. When a region is chosen, all the photos in your database that were taken in that region should be displayed (no sections are required). When a photo is then chosen, it should be displayed in the same way photos were displayed in last week's assignment.
5. All of your table views everywhere in your application (including the Recents tab) must be driven by Core Data (i.e. not `NSUserDefaults` nor Flickr dictionaries). You no longer have to support "pulling down to refresh" (though see Extra Credit 1).
6. You must use `UIManagedDocument` to store all of your Core Data information. In other words, you *cannot* use the `NSManagedObjectContext`-creating code from the demo.
7. Fetch the `URLforRecentGeoreferencedPhotos` from Flickr periodically (a few times an hour when your application is in the foreground and whenever the system will allow when it is in the background using the background fetching API in iOS). You must load this data into a Core Data database with whatever schema you feel you need to do the rest of this assignment.
8. Display a thumbnail image of a photo in any table view row that shows Flickr photo information. You must download these on demand only (i.e. do not ask for a thumbnail until the user tries to display that photo in a row). Once a thumbnail has been downloaded, you must store the thumbnail's data in Core Data (i.e. don't ask Flickr for it again). Don't forget that table view cells are reused!
9. Do not store Flickr photos themselves (i.e. any image other than a thumbnail) in your Core Data database. Fetch them from Flickr on demand each time.
10. Your application's main thread should never be blocked (e.g. all Flickr fetches must happen in a different thread). You can assume Core Data will not significantly block the main thread.
11. Your application must work in both portrait and landscape orientations on both the iPhone and the iPad and it must work on a real iOS device (not just the simulator).

Hints

1. Put your own Flickr API key into `FlickrAPIKey.h` or your queries will not work.
2. You will not be using `URLforTopPlaces` this time and you will not need `URLforPhotosInPlace:maxResults:`.
3. Since your user-interface is supposed to be essentially the same as Assignment 5, keep your storyboards from the last assignment (copy/paste their contents if you start with a fresh project) and also keep all of your table view classes (just gut them of all Flickr fetching and dictionary-accessing code and make them inherit from `CoreDataTableViewController`). At worst this will be a good starting point.
4. You can rename a class everywhere in an application by selecting that class name somewhere, then using the Rename… menu item in the Refactor menu inside the Edit menu in Xcode (since a lot of this application that is about Regions was about Places in last week's assignment, if you are trying to start from last week's code-base, you might want this sort of rename).
5. You will definitely want your table views to inherit from `CoreDataTableViewController` (provided). All that is needed to make this class work is to set its `fetchedResultsController` property (properly).
6. The fact that the `UIManagedDocument` opens/creates asynchronously has ramifications for your entire application design since its `NSManagedObjectContext` may not be ready the instant your application's UI appears. Design your code as if it might take 10 seconds or more to open/create the document (meanwhile your UI is up, but empty, which is fine). It never will actually take that long, but your code should work if it did.
7. Your schema needs to support the specific needs of your application. When you add a photo to the database, feel free to set attributes in Entities other than your Photo entity which can support the querying/sorting you need to do.
8. With the proper schema, this entire application can be built with very straightforward sort descriptors and predicates. **Put your brainpower into designing the right schema rather than building complicated predicates**.
9. For example, you'll likely want track directly the photographers who are active in a place or region (rather than relying on figuring it out from the photos all the time).
10. With the right schema, you should **not** need advanced KVC querying like `@count`. It's perfectly fine to add an attribute to any entity that keeps track of a count of objects in a relationship if you want.
11. Getting the place (and then region) where a photo was taken is quite a bit more interesting this time. When you download a photo dictionary from Flickr it does **not** include the region where the photo was taken (only the place the photo was taken in the form of a `place_id`).

12. To get the name of a place's region from a `place_id`, you will need to query Flickr *again* at the `URLforInformationAboutPlace:` (off the main queue, of course) using the `FLICKR_PHOTO_PLACE_ID` found in a photo dictionary and then pass the dictionary returned from that (converted from JSON) to `extractRegionNameFromPlaceInformation:`.
13. You can assume that all regions have unique names (i.e. you don't need to use the region's unique identifier).
14. You will probably want any fetches you execute on your Core Data database to ignore places/regions that you don't (yet) know the name of. If you never learn the name of the region a photo is in, then it's fine to treat it as if you never downloaded that photo.
15. Even though Extra Credit 1 is "extra credit," you might find hooking a Refresh Control up somewhat helpful in your debugging cycle. You can also force a fetch using "Simulate Background Fetch" in the Debug menu in Xcode.
16. The most important thing a table view connected to a Core Data database needs is an `NSManagedObjectContext`. You should think about how each table view Controller is going to acquire the context it needs. Each Controller might acquire it in a different way depending on how the Controller is used.
17. Only Controllers at the very top-level of your storyboard should need to get their contexts from the Application Delegate (via notification). Any Controller that is segued-to should get its context from the Controller that prepares it for that segue.
18. All `NSManagedObject` instances know the `NSManagedObjectContext` they are part of (they have a `@property` which returns it). You will likely find this useful (i.e. you'll find yourself needing a context in a certain situation and all you will have at hand is an `NSManagedObject` and that's all you'll need).
19. Don't forget that an `NSFetchedResultsController` will automatically update the table it's connected to any time the database context it is using to watch the database changes in any way that would affect what is being shown. This is awesome. As long as you use the same `NSManagedObjectContext` throughout your entire application (highly recommended), tables will always stay in sync no matter what changes you make to the database.
20. You can limit the number of results an `NSFetchResult` fetches with its `fetchLimit` `@property`. However, for an `NSFetchedResultsController` this only works when it explicitly performs its fetch (in other words, the `fetchLimit` will not continue to be enforced as the `NSFetchedResultsController` "watches" the context for changes). Invoking `performFetch` in `CoreDataTableViewController` will cause its `NSFetchedResultsController` to fetch again (and respect the `fetchLimit`). How often should you do this (i.e. what does "occasionally" mean)? That's part of what this assignment is asking you to figure out. There are multiple mechanisms to do something "occasionally" in your table view (and the best solutions will probably be tied to what's going on with the database somehow).

21. The amount of code in each of your table view Controllers should be relatively small. This is because `NSFetchedResultsController` is going to be doing all the work. Let it. If you find yourself lots and lots of code in each table view Controller then either you are not using object-oriented programming to its best advantage, or you are entirely on the wrong track (or both). Each Controller really just needs to …
    a. acquire an `NSManagedObjectContext` from an appropriate place
    b. create an `NSFetchedResultsController`
    c. implement `cellForRowAtIndexPath:`
    d. (possibly) support navigation from itself

    Inheritance among Controllers is also quite useful (as you've hopefully already seen).
22. You will very likely want to continue to do all of your fetching (both photos and now place information) in your Application Delegate (except possibly for Extra Credit 1) and let the rest of your application simply enjoy the fruits of your Application Delegate's labor.
23. You are welcome (encouraged) to use the code from the in-class demos, but you will learn much less by copying/pasting code (and then modifying it) than you will by typing it in from scratch. Having said that, the infrastructure in the Application Delegate can pretty much be used as-is (except that you must get your context from a `UIManagedDocument`).
24. For background fetching to work, remember that you must edit your project settings and turn Background fetch on in the Capabilities section under Background Modes.
25. Setting the thumbnail image in a `UITableViewCell` is as simple as `cell.imageView.image = …` And if this is the first time the user has seen this row in a table, you'll have to fetch it asynchronously from Flickr. This will bend your mind a bit when you factor in that the table view might be in the middle of scrolling and it reuses cells! It's not difficult coding-wise, but it requires you to use multithreading and to think about the threads of execution in your application carefully. Imagine that fetching the thumbnail image from Flickr takes 10 seconds or more and that the user is merrily scrolling up and down in the table all that time. What are you going to do when the image data finally comes back?

Things to Learn

Here is a partial list of concepts this assignment is intended to let you gain practice with or otherwise demonstrate your knowledge of.

1. Core Data
2. `UIManagedDocument`
3. Database Schema Design
4. Multithreaded Application Development
5. The Application Delegate method `application:didFinishLaunchingWithOptions:`
6. Background Fetching
7. Background URL Sessions
8. Listening to "radio stations" via `NSNotificationCenter`

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.
- Main thread blocked waiting for network I/O.
- Bad database schema design leading to tortuous code.

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

1. Still allow the user to force a fetch using the `refreshControl` in the appropriate table views. And use the cellular network to fetch *only* when the user prompts you via a `refreshControl` in this manner (otherwise restrict all your `URLforRecentGeoreferencedPhotos` fetches to WiFi only). If you are a bit stumped about how to know when to `endRefreshing` in a table view, think about whether there are some `NSNotification`s flying around that you could listen in on.
2. Loading Flickr information into your database can be ridiculously inefficient if, for each photo you download, you query to see if it is already in the database, add it if it is not (and maybe update it if it is). Enhance your application to make this download much more efficient, preferably only doing two or three queries *total* in the database for each "batch" of Flickr photos you download (instead of hundreds, which is what Photomania does). The predicate operator `IN` might be of value here.
3. If you were to use your application for weeks, your database would start to get huge! Implement a mechanism for pruning your database over time. For example, you might want to delete photos a week after you download them?
4. Teach yourself how to use `NSFileManager` and use it along with `NSData`'s `writeToURL:atomically:` method to cache image data from Flickr (the photos themselves) into files in your application's sandbox (you'll probably want to use the `NSCachesDirectory` to store them). Maybe you keep the last 20 photos the user looks at or all the photos in Recents or maybe 50MB's worth of photos? Up to you.
5. Adding the fetch for place and region information is going to break the way background fetching updates the app-switcher. Can you figure out why? Fix this problem. Hint: It is the background fetch completion handler that tells iOS to update the UI in the app-switcher. You can simulate a background fetch while running in the Simulator using the "Simulate Background Fetch" menu item in Xcode's Debug menu.
6. Switching to `UIManagedDocument` for our `NSManagedObjectContext` is going to break launching due to a background fetch event. Can you figure out why that is? Fix it. Remember that you can launch your application as if it were happening due to a background fetch by choosing Edit Scheme from the popup in the upper left corner of Xcode (near the run and stop buttons), then turning on "Launch due to a background fetch event" in the Options tab of the window that appears. You can also create a new scheme there that has that switch always on for easy background fetch launching during debugging.