Assignment VI: CoreData 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 primary work to be done in this assignment is to improve the performance of the application by using a database to cache the information from Flickr, and spruce up the user-interface with some thumbnail images.

Improving performance is definitely an aspect of this assignment (we are doing so by minimizing our chatter with Flickr, primarily through caching), but your application’s user-interface will still be blocked on those occasions when you are forced to talk to Flickr. We’ll fix that problem next week by introducing threads.

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. You will 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).

Be sure to check out the Hints section below!

Also, check out the latest in the Evaluation section to make sure you understand what you are going to be evaluated on with this assignment.

Materials

• The class CoreDataTableViewCellController is provided.
• You will still need your Flickr API key.


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 your Core Data database to present it.

3. Present a “Refresh” button in an appropriate spot in your user-interface which queries Flickr to get the latest list of SPoT photos and updates your database. The effect of this button should happen automatically if your database is empty when your application starts up. Otherwise your application should not query Flickr except to get photo/thumbnail image data (and then only on an as-needed basis since image data will be cached (see below)).

4. 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. Never ask Flickr for the same thumbnail twice (cache the thumbnails’ data in your Core Data database).

5. Cache (non-thumbnail) photo images viewed by the user into files in your application’s sandbox. Each photo’s image should be a separate file in the sandbox. Do not cache photo image data in Core Data (thumbnails are the only images whose data goes there). Limit the cache to about 10MB total. When this limit is reached, the oldest photos in the cache should be evicted (deleted) to make room for new photos coming in. Your application should never query Flickr for the image data for a photo that it has in its cache (obviously). This cache should persist between application launches.

6. Keep as little of the photos’ image data in memory as you can (i.e. let the associated NSData and/or UIImage objects get deallocated when you are done using them). You should have at most 2 photos’ image data in memory at a given time, preferably only 1 (or even 0 if none are being viewed).

7. Sort all tables alphabetically (and case insensitively) except the Recents table which should be sorted with the most recently viewed first. Only show Recents that have been viewed within 24 hours of the last time the Recents list appeared on screen and don’t show more than 20 recents. Note: If you have done all of the above Required Tasks properly, then browsing through the most-recently-viewed 10MB worth of photos should not require any network activity.
Hints

1. When you create your New Project ... for this assignment, be sure to click the box that says “Use Core Data”. This will give you the very important application delegate property `managedObjectContext` which you will need to use when fetching existing objects from your database using an `NSFetchRequest` or to creating new objects in your database using `NSEntityDescription`. It will also create a blank object database model for you.

2. You can get/set your Entities’ Attributes using `valueForKey:`/`setValueForKey:` if you prefer, but, especially when it comes to getting images (and loading up the database too), 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.

3. 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).

4. Do not use an attribute in your data model named `id`. You can imagine the problems this might cause in Objective-C.

5. 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.

6. 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 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).

7. Make sure you save changes you make to your database (at appropriate times) using the `save:` method in `NSManagedObjectContext`.

8. 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:`.
9. Note that if you have a two-way relationship in your model, setting one side automatically sets the other side properly for you.

10. “To many” relationships are represented simply as an \texttt{NSSet} of \texttt{NSManagedObject} instances. So to set the value of a property that is a to-many relationship, just create an \texttt{NSMutableSet}, add \texttt{NSManagedObject} instances to it, then set the property to that \texttt{NSMutableSet}.

11. You do not have to use the provided \texttt{CoreDataTableViewController} for this assignment, but it’s extremely likely you’ll want to! Its implementation is mostly just copy/pasted from the documentation of \texttt{NSFetchedResultsController}. It’s very easy to subclass and use. The most important thing you must do is set its \texttt{fetchedResultsController} property. Outside of that, you’re going to need to implement the delegate method that does something when a row is selected and also override the appropriate delegate method to make the cells display the proper information from the managed objects in the table.

12. 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 \texttt{CoreDataTableViewController} do all the work for you.

13. \texttt{NSFetchedResultsController} has a property called \texttt{fetchedObjects} which is the array of \texttt{NSManagedObject}s it fetched. Checking whether this array is empty (empty, not \texttt{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 (part of Required Task #3). If \texttt{fetchedObjects} is \texttt{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 \texttt{super}'s implementation so that \texttt{CoreDataTableViewController} can do what it wants to do too.

14. The sorting in \texttt{CoreDataTableViewController}-based view controllers should happen automatically as part of the \texttt{NSFetchRequest} you use to create your \texttt{NSFetchedResultsController}. Just make sure it has the right \texttt{NSSortDescriptor}.

15. Both the \texttt{predicate} and the \texttt{sortDescriptors} for your Recents \texttt{CoreDataTableViewController}'s \texttt{NSFetchedResultsController}'s \texttt{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.

16. Your Recents \texttt{CoreDataTableViewController}'s \texttt{NSFetchedResultsController}'s \texttt{NSFetchRequest} probably changes every time the Recents appear on screen (read Required Task #7 carefully). You \textbf{cannot} just modify the \texttt{predicate} in your controller’s existing \texttt{NSFetchedResultsController}'s \texttt{NSFetchRequest}, you need to create a \textbf{new} \texttt{NSFetchedResultsController} each time you change anything about the \texttt{NSFetchRequest} that your \texttt{CoreDataTableViewController}’s
NSFetchedResultsController is using. And again, if you are doing this in one of the View Controller’s lifecycle methods, be sure to let super do its job at the right time. CoreDataTableViewController re-fetches its data whenever its NSFetchedResultsController changes (though for performance reasons it will defer that fetch to viewWillAppear: if the view is not on screen at the time the NSFetchedResultsController changes).

17. Note that whenever you save: your Core Data database (e.g. after adding a thumbnail image’s data to it or changing a last-viewed property or something), things will automatically update in the user-interface without your having to do anything. That is the wonder of NSFetchedResultsController and Core Data’s use of the key-value observing mechanism. However, if you do not save: when changes are made, things will not update (nor will your changes persist between runnings of your application).

18. The FlickrFetcher method urlStringForPhotoWithFlickrInfo: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). You’re going to want to store both of these urls into your database.

19. [NSFileManager defaultManager] can be used to create directories, find out if files exist, delete files, etc. Part of the intent of this assignment is for you to read through the documentation for this class and figure out how to use it.

20. Be careful to pick the appropriate directory from NSSearchPathDirectory to use to store your cached photo image data. Also, if you want to clear your sandbox entirely, delete your application (see Hint #6 above).
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.).
- Assignment was turned in late (you get 3 late days per quarter, so use them wisely).
- Code is too lightly or too heavily commented.
- Code crashes.
- Code leaks memory!
- User-interface hangs or is blocked from action by the user.
Extra Credit

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

1. Make your application work on the iPad with appropriate user-interface idioms on that platform (i.e. on the iPad, you do not have to use a UINavigationController for all three view controllers).

2. Make one or more of your table views searchable. There is a public property in CoreDataTableViewController that lets you specify a key in the NSManagedObjects which are displayed in the table to search on. Give it a try. It’s a one-liner.

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 it be divided into sections (alphabetical). The simplest way to do 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?

4. Allow users to delete photos (i.e. they no longer appear in tables). For a real challenge, do it in a way that makes it so that the Refresh button does not bring them back!