Assignment VI: Virtual Vacation

Objective

In this series of assignments, you have been creating an application that lets you browse photos posted on Flickr. In this final installment, you will use what you have built so far to allow your user to build a "virtual vacation" by visiting photos from around the world.

The primary work to be done in this assignment is to use CoreData to build a database of photos organized by vacation which the user can peruse and search.

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 `CoreDataTableViewController` is provided. A new version of `FlickrFetcher` is also available and is necessary for this assignment.
- You will still need your Flickr API key.

Required Tasks

This application lets the user assemble a Virtual Vacation of places in the world to visit. Users will use the photo-choosing capabilities of your Fast Map Places application to choose photos in the places they want to go. In this application you will have two major tasks: allowing the user to choose where they want to go and allowing the user to "go on vacation" in their Virtual Vacation spots. You will accomplish the former by adding a "Visit/Unvisit" button to the scenes in your Fast Map Places where a photo is displayed. You will accomplish the latter by adding a new tab to your Tab Bar Controller which lets the user peruse their Virtual Vacation either by place or by searching for tags that were found in the Flickr dictionaries for the photos they chose to visit.

1. Add a new tab to your application that displays a new table view controller showing a list of all the "Virtual Vacations" found in the user's `Documents` directory in their sandbox. A Virtual Vacation file is created by saving a `UIManagedDocument` (more on this below). Each vacation must have it's own separate file in the `Documents` directory.
2. When the user chooses a Virtual Vacation from the list, bring up a static table view with two choices: Itinerary and Tag Search.
3. The Itinerary tab must show a list of all the places where photos in the chosen Virtual Vacation have been taken (sorted with first-visited first). Clicking on a place will show all the photos in the Virtual Vacation taken in that place. The place name should be the one returned by getting the new `FLICKR_PHOTO_PLACE_NAME` key in the Flickr photo dictionaries you retrieve from the `photosInPlace:maxResults:` method. You will need to use the new `FlickrFetcher` code available with this assignment. Use only the place's name (as returned the the `FLICKR_PHOTO_PLACE_NAME` key) to determine what the place is (i.e. ignore things like the Flickr place id).
4. The Tag Search tab must bring up a list of all the tags found in all the photos that the user has chosen to be a part of this Virtual Vacation (sorted with most-often-seen first). Touching on a tag brings up a list of all the photos in the Virtual Vacation that have that tag. The tags in a Flickr photo dictionary are contained in a single, space-separated, all lowercase string (accessible via the `FLICKR_TAGS` key). Separate out and capitalize each tag (not all caps, just capitalize the first letter) so that they look nicer in the UI. Don't include any tags that have a colon in them.
5. You are not required to provide any UI for users to create new Virtual Vacations (see Extra Credit #1 if you want to), so simply create a single Virtual Vacation called "My Vacation" in the user's `Documents` directory somewhere in your code. All "visits" and "unvisits" will happen in this "My Vacation" Virtual Vacation. However, just because you do not have to provide UI to create new vacations does not mean you are exempted from any other required tasks which support multiple Virtual Vacations (like Required Task #1). You may well want to create some other Virtual Vacations anyway just to verify that the rest of your code deals properly with multiple of them.

6. All of the new tables in this application that show places or photos or tags must be driven by Core Data. It is your responsibility to determine the schema which best supports your needs.

7. To make all this work, of course, you will need to add a Visit/Unvisit button to the scenes in the storyboard which show a photo. If that photo is already in "My Vacation," then the button's title should appear as "Unvisit," otherwise it should appear as "Visit." Clicking this button toggles whether the photo is part of "My Vacation" or not.

8. You do not have to get this working on both platforms. Pick whichever of the two you want. It does, however, have to work on a real device, so pick the device you have the hardware for.

Hints

1. It is much, much easier if you only ever have one `UIManagedDocument` instance per document-on-disk (remember that each vacation is its own document on disk), which is shared by your whole application (though it is perfectly legal to open the same document multiple times each with its own instance of `UIManagedDocument`).

   There are a number of ways to share a `UIManagedDocument` instance for each file. For example, you could have a "helper" class that only has a single class method to hand out a shared `UIManagedDocument` for a given vacation:

   ```
   @interface VacationHelper
   + (UIManagedDocument *)sharedManagedDocumentForVacation:(NSString *)vacationName;
   @end
   ```

   In this case, though, anywhere your code uses a document, it would have to make sure it was properly opened or created first. This could result in a fair amount of duplicated code.

   Or you could use an API like this to have the helper class open or create the document as needed before handing it out (but since opening and creating are asynchronous, this helper method would have to hand the `UIManagedDocument` to the caller using a block).

   ```
   typedef void (^completion_block_t)(UIManagedDocument *vacation);

   @interface VacationHelper : NSObject

   + (void)openVacation:(NSString *)vacationName
             usingBlock:(completion_block_t)completionBlock;

   @end
   ```

   It is up to you how you want to share `UIManagedDocuments` for a given vacation, but it is highly recommended that you do so.

2. If you do not follow Hint #1 and end up with multiple `UIManagedDocument` instances for the same document, you must keep them in sync yourself. This will not happen automatically. That's why we want all code that is looking at a particular document to be using the same instance of `UIManagedDocument`.

3. If you are using the second approach in Hint #1, check the lecture on blocks to refresh your memory how to invoke a block (like `completionBlock`) from your code.

4. You might find the `NSFileManager` method `contentsOfDirectoryAtURL:includingPropertiesForKeys:options:error:` useful for finding out what vacations the user has (even though you're only going to actually

create one vacation, you have to write the code as if you support any number of them).

5. Give some thought to how you will pass the user's chosen vacation (from the table in Required Task #1) around. For example, your static table view will likely have to pass that information on through to the controllers it segues to.
6. Remember that all `NSManagedObjects` know what `NSManagedObjectContext` they are in (can be accessed via the `NSManagedObject`'s `managedObjectContext` property).
7. `NSManagedObjectContext` is **not** thread-safe! Only access an `NSManagedObjectContext` in the thread in which it (or its `UIManagedDocument`) was created or use the `performBlock:` or `performBlockAndWait:` method in `NSManagedObjectContext` (which will make sure the block is performed on the right thread for that context, which may well be the main thread). You do not have to do any of this for this assignment (just use the main thread for all Core Data access).
8. The string `@"My Vacation"` should only appear on one line of code in your application (somewhere in your Visit button implementation). In other words, don't hardwire it anywhere else; the rest of your implementation should be generic with respect to vacation documents.
9. The table of places in a vacation has to be ordered by first-visited first, so you will need to have an attribute that records when this place was first added to the database.
10. You will also need an attribute somewhere in your database that keeps track of how many photos have a given tag (it's easy to calculate if you have a to-many relationship between these search tags and the photos which have those tags). You'll have to update this attribute both when new photos are added to the vacation and when photos are deleted ("unvisited").
11. The list of photos in a place in the vacation can be ordered however you want.
12. You can get/set your Entities' Attributes using `valueForKey:/setValueForKey:` if you prefer, but 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 as described in lecture. This is especially true when it comes to creating objects (since you'll want to properly set up any relationships between the objects) and deleting objects (since you'll want to clean up any relationships between objects in `prepareForDeletion`).
13. When you insert an object in the database, that's a good time to insert any other objects it depends on too and set up the relationships between them (simply by assigning values to the properties involved). The only modification this application makes to the database is to add a photo (by "visiting" it), but obviously many other objects may be created in the database when that happens. Collecting all that database modification into the method that adds the photo would be pretty clean.

14. When you delete an object from the database, be careful not to hang onto a `strong` pointer to it (since it will be gone after you delete it).
15. 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).
16. Do not use an attribute in your data model named `id`. You can imagine the problems this might cause in Objective-C.
17. `NSLog()`ing `NSManagedObjects` is very useful. It will show you lots of detail (assuming the object in question has already been faulted in from the database).
18. 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 using `NSFetchedResultsController` if there is no Entity that represents a "photo" in the database.
19. If you change your object model (and you will likely do that numerous times as you iterate on your schema), be sure to <u>delete your application from your device or simulator</u> before running it with a new schema so that your old database (with the old schema) gets removed. 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).
20. Make sure you save changes you make to a vacation document using `saveToURL:forSaveOperation:completionHandler:`. You will probably only be modifying your document in the implementation of your Visit button.
21. Note that if you have a two-way relationship in your model, setting one side automatically sets the other side properly for you.
22. "To many" relationships are represented simply as an `NSSet` of `NSManagedObject` instances (or subclasses thereof). You can make a `mutableCopy` of the `NSSet` and manipulate it (then set the property to your mutable copy) or there are "convenience methods" to add/remove objects from the set if you make a custom subclass of your `NSManagedObjects` (which is highly recommended).
23. 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 only thing you have to do to make it work is to set the `fetchedResultsController` property. The only `UITableViewDataSource` method you're going to *have* to implement is `tableView:cellForRowAtIndexPath:`. You'll also want to implement `tableView:didSelectRowAtIndexPath:` and/or

`prepareForSegue:sender:`. Using the method `objectAtIndexPath:` on `self.fetchedResultsController` is wonderful for implementing these.

24. Note that whenever you save your Core Data database (by saving your vacation document), things will automatically update in the user-interface without your having to do anything if you use `NSFetchedResultsController` (like `CoreDataTableViewController` does). That is the wonder of `NSFetchedResultsController` and Core Data's use of the key-value observing mechanism. If you implement your table views without using `NSFetchedResultsController` (e.g. without `CoreDataTableViewController`), then you'll have to figure out how to get your tables updated on your own.

25. 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` (via `CoreDataTableViewController`) do all the work for you.

26. The sorting in `CoreDataTableViewController`-based view controllers will happen automatically as part of the `NSFetchRequest` you use to create your `NSFetchedResultsController`. Just make sure each has the right `NSSortDescriptor`(s) (and the right attributes in your database schema).

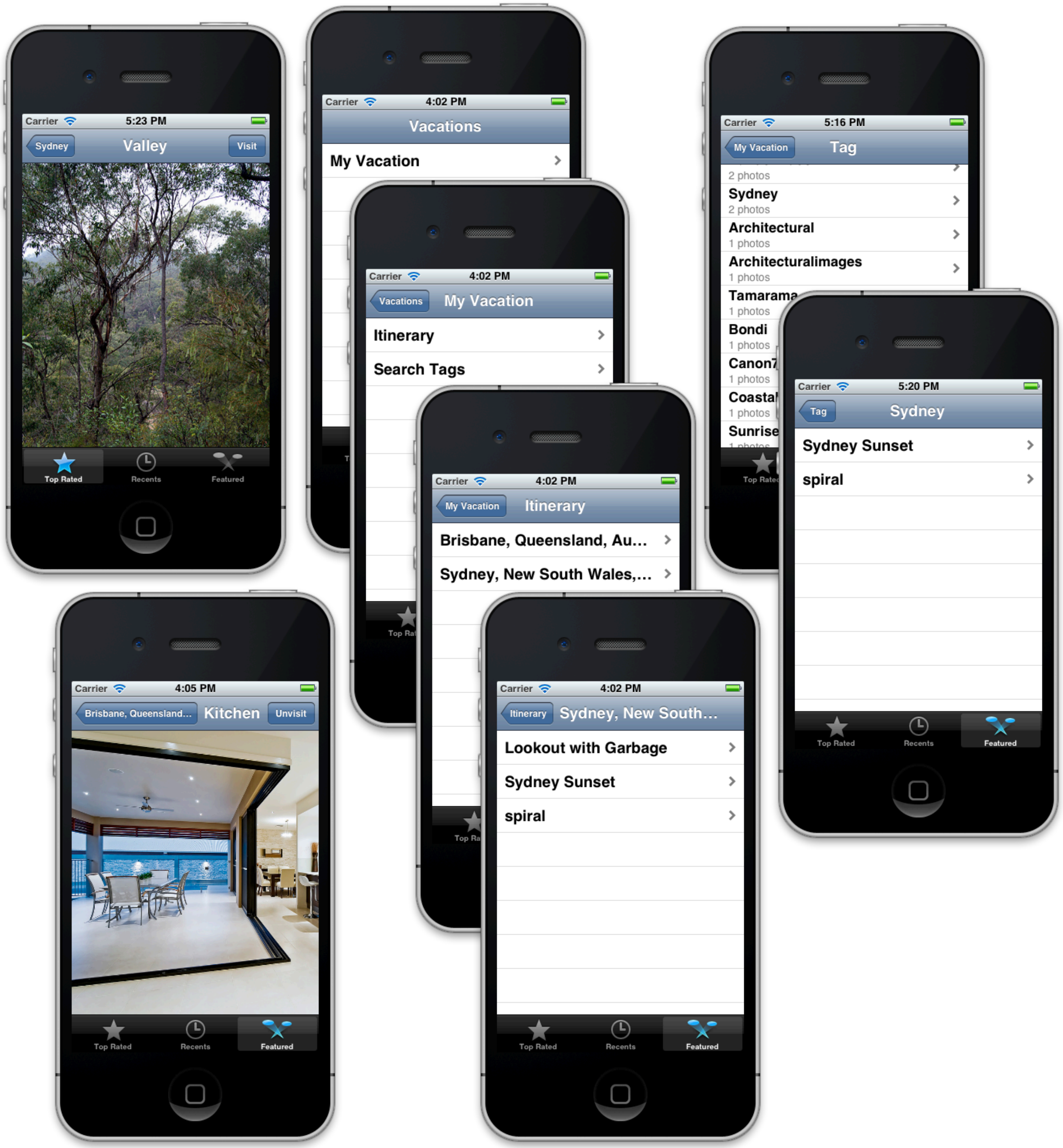
27. You do not have to update your Recents tab when users are "on vacation." In other words, whatever code you have there from the last assignment is fine.

28. Your image-displaying MVC from the last assignment needs a couple of upgrades. It must now be able to show an image which is specified by a photo in the database (for when the user is "on vacation") in addition to one defined by a Flickr photo dictionary (your existing use of it) and it must be able to support scenes that include the Visit button. It is strongly recommended that you use good object-oriented programming technique here. For example, you may well want a class whose entire implementation is just the handling of the Visit button (and which inherits all the rest of the behavior it needs to display the photo).

29. Be careful of the case where you hit the Univisit button while the user is viewing the photo while "on vacation." You might have to clear the image view on the iPad because you'll have deleted the photo from the database and thus can't Visit it again. That's a fine solution. And on the iPhone, you'll probably want to pop your navigation controller in that case.

Screen Shots

You don't have to implement your user-interface exactly like this, but you must fulfill all required tasks.



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.
- 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. Add UI to allow the user to add new vacation documents and to pick which vacation they are visiting when they choose the Visit button.
2. Allow users to reorder their itinerary for their vacation. To do this, you might want to think about creating a top-level Entity (Itinerary) in your schema and using an "ordered to-many relationship" to store the places in the itinerary. An "ordered to-many relationship" appears in your code as an `NSOrderedSet` (instead of an `NSSet`). The table view that shows the places in the itinerary will have to be rewritten to display this `NSOrderedSet` of places (it won't be able to be an `NSFetchedResultsController`-based table view) and you will have to figure out how to use `UITableView` API to edit an `NSMutableOrderedSet`. Warning: while this is not that difficult to implement coding-wise, it requires quite a bit of investigation to figure out. Another approach is to add an attribute in your schema that determines the order (but this can be a little bit clunky). The former approach will probably lead to more learning opportunities.
3. Let the user search inside the tags table view. This is a non-trivial extra credit exercise, but you'll definitely want to use `NSSearchBarController`.