Paparazzi - Part 2

Due Date

This assignment is due by **11:59 PM, February 10.**

Assignment

Last week, you created the first version of Paparazzi, an iPhone application for viewing online photos. This week, we'll improve the application in two major ways. First, we'll make the switch from hardcoded views to table views, so that our application can display large, dynamic data sets. Second, we're going to move from using property lists to CoreData for local storage so we'll have a foundation for storing and searching large amounts of data.

Here are the requirements for Part 2:

1. **Read the contents of FakeData.plist at an appropriate place in your application's life cycle and store it into a CoreData database**. If you haven't already you'll need to add the file to your Xcode project so that it's built into the application bundle. CoreData's SQLite backing is persistent, so the code shouldn't duplicate data on subsequent launches. Handle common error conditions gracefully.
2. **Define Person and Photo model objects.** A Person should include a name, and a set of photos. Each Photo should link to a Person, and should include a name and a path. So, for each item in the property list above, you'll need to instantiate a Photo object and associate it with a corresponding Person object. The walkthrough covers this in more depth.
3. Update the PersonListViewController and PhotoListViewController classes to **manage a plain style UITableView**. You may want to make them into subclasses of UITableViewController. **Display the list of Person objects in the table view**, including an image and a name. Use a corresponding NSFetchedResultsController to manage objects fetched from the CoreData store.
4. **When a person is selected in the list**, set up and push a PhotoListViewController
5. **When a photo is selected in the list**, set up and push a PhotoDetailViewController
6. The PhotoDetailViewController class needs to be updated as well. It should have a photo property which a client can set. It will **manage a UIScrollView** where it **displays a photo and allows the user to zoom in and out**.

There is an archive accompanying this assignment titled **Paparazzi2Files.zip** which includes the FakeData.plist file that your application will read. Additionally, it includes a class called FlickrFetcher which facilitates reading from the CoreData store.

Testing

In most assignments testing of the resulting application is the primary objective. In this case, testing/grading will be done both on the behavior of the application, and also on the code.

We will be looking at the following:

1. Your project should build without errors or warnings and run without crashing.
2. Each view controller should be the File's Owner of its own Interface Builder document. **Do not put your entire application into a single Interface Builder document!**
3. You should be using retain, release and autorelease correctly at this point. **Don't leak memory or over-release**.
4. Since the project is getting more complex, **readability is important**. Add comments as appropriate, use descriptive variable and method names, and decompose your code.
5. Your program should behave as described above, reading in a photos from the property list, building a CoreData database, listing users and photos in a table view, and displaying and manipulating a photo when selected.

Walkthrough

Reading the FakeData.plist file
You may first want to inspect the contents of the property list in a text editor. Once you're familiar with the structure, you'll want to use the NSBundle class to get the path for the property list. Many of the Foundation classes have constructors that let you specify a path- in this case, you'll probably want to use +[NSArray arrayWithContentsOfFile:].

The Photo model object
It's pretty clear at this point that **we need a model object to package together all the bits of data relating to a photo.** At the bare minimum, it should keep track of an image (or image URL), the name of the photo, and a link to a person.

The Person model object
Each Photo must be associated with a Person. Your Person object should at least have a user name, and may have taken multiple photos. Keep this in mind as you build your CoreData model graph.

Populating your database
The accompanying **Paparazzi2Files.zip archive includes a class called FlickrFetcher.** Here is the class definition:

```
@interface TwitterHelper : NSObject {
    NSManagedObjectModel *managedObjectModel;
    NSManagedObjectContext *managedObjectContext;
    NSPersistentStoreCoordinator *persistentStoreCoordinator;
}

// Returns the 'singleton' instance of this class
+ (id)sharedInstance;

// Checks to see if any database exists on disk
- (BOOL)databaseExists;

// Returns the NSManagedObjectContext for inserting and fetching
objects into the store
- (NSManagedObjectContext *)managedObjectContext;

// Returns an array of objects already in the database for the
given Entity Name and Predicate
- (NSArray *)fetchManagedObjectsForEntity:(NSString*)entityName
withPredicate:(NSPredicate*)predicate;

// Returns an NSFetchedResultsController for a given Entity Name
and Predicate
- (NSFetchedResultsController *)
fetchedResultsControllerForEntity:(NSString*)entityName
withPredicate:(NSPredicate*)predicate;
```

When only one instance of a class is used across an application's execution, it is referred to as a "Singleton". The singleton for the **FlickrFetcher** class can be accessed using the **+[FlickrFetcher sharedInstance] method**. Once you have the singleton, you can call any of the accompaniying instance methods.

To check if a database exists for your app, use the **-[FlickrFetcher databaseExists] method.** This will be helpful in deciding whether or not you need to parse the plist and populate your CoreData store when you launch your application. If you need to clear out a database from a previous launch, you can **tap and hold on the application icon on your iPhone/iPod touch** to delete the app – this will delete all content as well so you can run fresh. You can also select "**Reset Content and Settings..." from the "iPhone Simulator" menu** to start from scratch. Be sure to populate your database as early as you can in your app's execution, since **calling any of our CoreData helper methods will create the database.**

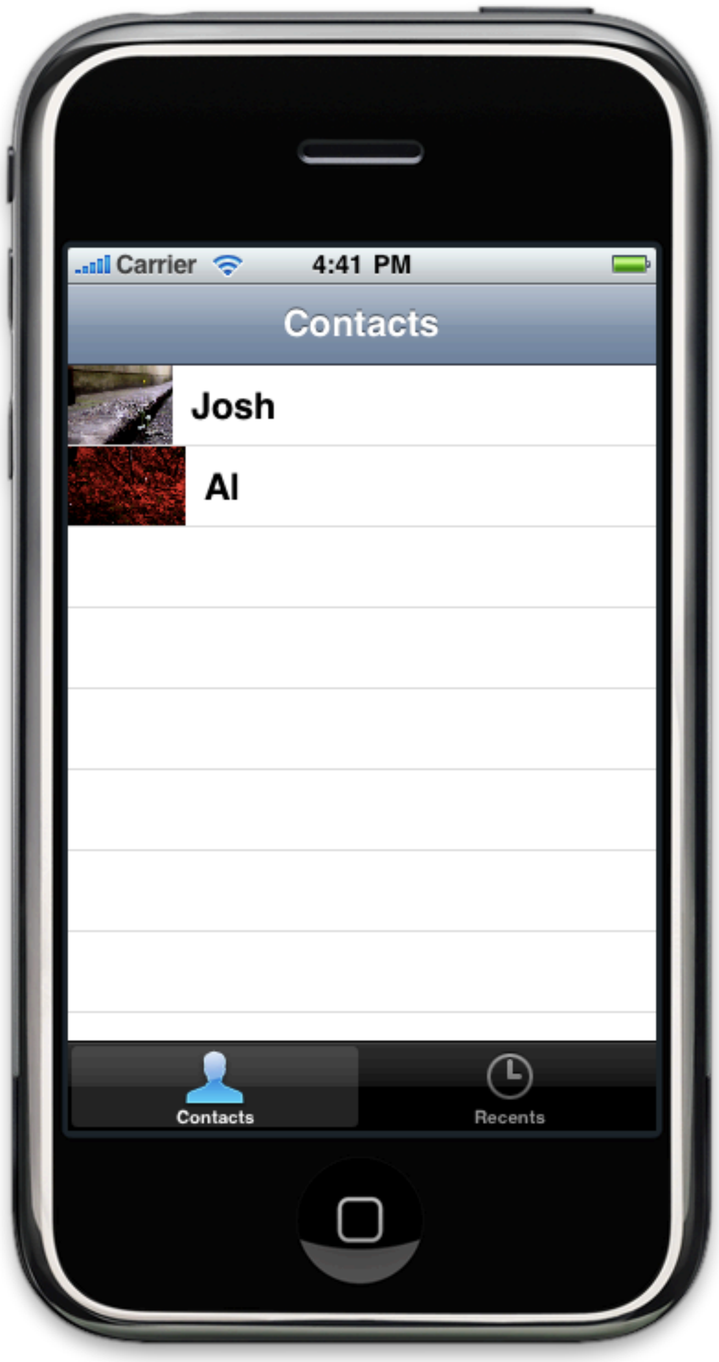
When populating your database, **you'll need to use a NSManagedObjectContext to insert new Entities.** You can get this using **-[FlickrFetcher managedObjectContext].** If you need to, you can use the **[FlickrFetcher fetchManagedObjectsForEntity:withPredicate] method to fetch existing objects from your CoreData store.**

Filling in your Table Views

When filling in your Table Views, take a close look at the **NSFetchedResultsController class**. This class uses similar conventions to UITableView, and will prove very useful in hooking your Table Views to your database. You can use **-[FlickrFetcher fetchedResultsControllerForEntity:withPredicate:]** to build an appropriate Fetched Results Controller for your database for each of your Table Views.

Drag the header and source files for the FlickrFetcher class into your Xcode project. Do the same with the FakeData.plist file. For now, you don't need to know anything about what the FlickrFetcher class doing under the hood, but please feel free to look at the class to understand how these fetches are made on the database.

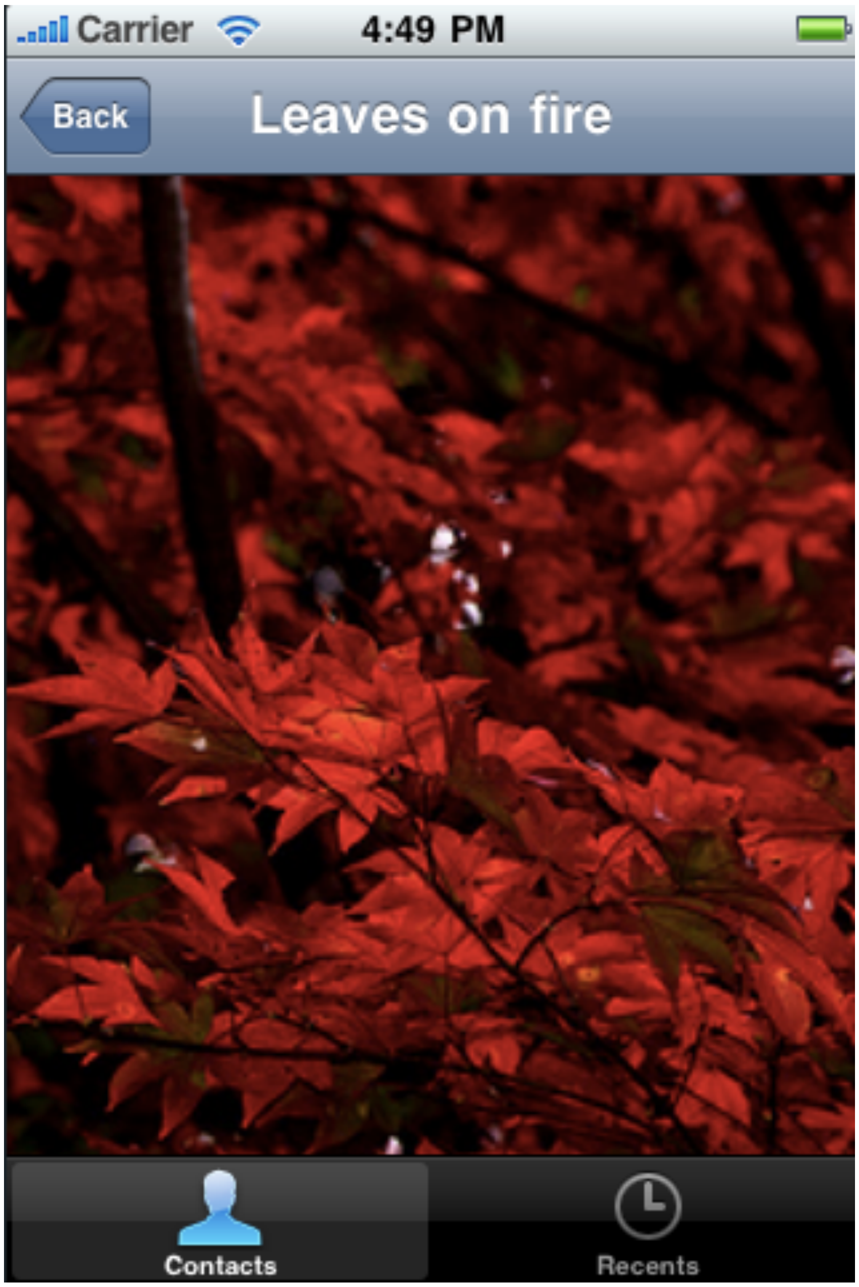
Managing a table view with PersonListViewController
You will probably want to use UITableViewController as the starting point for your view controller subclass. It automatically creates a table view with itself as the delegate and datasource, among other things. You don't even need to use a NIB if the table view is all you're displaying- just instantiate your subclass using -initWithStyle:. If you do choose to use a NIB and -initWithNibName:bundle:, be certain that your view outlet is pointing at a valid UITableView.



Responding to a selection
In Paparazzi 1, we pushed a PhotoListViewController onto the navigation stack in our button's action method. Now, we're going to do it in response to a table selection. You remember how to do that, right? Your PhotoListViewController class should now have a "person" property. Be sure to set it on the instance after creation, before you push it onto the stack.

Scrolling and zooming your photos

You're not done yet. Don't forget to add a scroll view to your PhotoDetailViewController to allow your user to pan and zoom around each photo. **Take a look at the UIScrollView delegate methods** to get your scroll view hooked up for zooming (and remember, by default, a scroll view only scales from 1.0 to 1.0)!



Extra Credit

Here are some suggestions for enhancing the second version of Paparazzi.

• Sort your list of people alphabetically
• Add support for reordering and deleting users from the table view. Use the view controller's standard Edit/Done button as described in Lecture 7, and implement the required datasource & delegate methods for allowing rows to be reordered and deleted. Consider how you may need to modify your CoreData model to support ordering.
• Customize your table view cells even further - display a timestamp for each photo, or a total count of photos for each user (or anything else you like).
• Add a "description" view that overlays on top of your photo, that does not scroll or zoom with the photo, but tells you the name of the user and title of the photo. Have the description disappear and reappear when you tap the screen.

If you undertake any extra credit, please let us know in your submission notes or otherwise. If you don't, we might not know to look for it. And be sure that the core functionality of your application is solid before working on any of this!