Assignment V:

Smashtag Mention Popularity

Objective

In this assignment, you will enhance the Smashtag application even further to do some analysis on all of the mentions in a search result using Core Data.

Submit your solution via the normal process before the start of lecture on Wednesday.

Be sure to review 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

- You will need to have successfully completed the previous assignment to do this assignment.
- You will probably want to use the FetchedResultsTableViewController from the lecture demo.

Required Tasks

1. In your Recent Searches tab, add a Detail Disclosure button  to every row. Touching it must segue to a new table view MVC which lists the most popular user and hashtag mentions (uniqued, case-insensitively) found in all tweets ever fetched for the search term in that row. This new MVC must also allow clicking on a row to segue to your original tweet-searching MVC.
2. This mentions popularity table should be sorted in order from the most popular mention to the least popular.
3. If two (or more) mentions have the same popularity, those mentions should appear in the table in alphabetical order.
4. Each row in the table should show not only the mention itself, but also the number of tweets found (by the given search term) in which that mention appeared.
5. For a mention to be considered "popular", it must have been mentioned in more than one tweet.
6. Once the mentions in a tweet have been counted, they should not be counted again if the same tweet is fetched again. In other words, don't "double-count" mentions in tweets that are fetched more than once.
7. All of the data which drives this new MVC must be stored in Core Data (and persist across launchings of your application).
8. The contents of this new MVC must be managed by an `NSFetchedResultsController` (you may use `FetchedResultsTableViewController` as your new MVC's Controller's superclass if you wish).
9. Don't break any of the rest of the functionality in your Smashtag application. There is no requirement that the MVCs you already have from the previous assignment be converted to use Core Data. Only your new MVC has to do this. Of course you'll need to modify your existing code to *collect* the data this new MVC is going to use.
10. You must not block the main thread of your application at any time. You can assume that any Core Data *queries* you make will not take so long that your main thread will be blocked (hopefully you design your schema such that this is actually true, even for very large numbers of downloaded tweets). But when you *load* data into your database, you should do that off of the main queue.
11. Your application must work properly in portrait and landscape on any iPhone (this is an iPhone-only application).
12. You must get this assignment working on an actual iOS device (not just the simulator).

Hints

1. If you copy/paste from another application's `AppDelegate`, be sure the name of your CoreData model file is properly set in that code.
2. A way to check that you have satisfied Required Task 6 is to search for a term, check the mention popularity, then search for it again (not refreshing it, but typing it in again) and making sure that the popularities don't change (or at least, only change by newly-tweeted tweets). If the popularity counts double, you know it's not working.
3. Nowhere in the Required Tasks are you required to display the full contents of Tweets or Users in your new MVC, so there's no need to build a huge schema that stores all the data that is fetched from Twitter. Just store the stuff you need to make your new MVC work.
4. With the proper schema, this entire application can be built with very straightforward sort descriptors and predicates (i.e. you will not need function predicates or compound predicates). **Put your brainpower into designing the right schema rather than trying to build complicated predicates**.
5. Don't limit your idea of what the entities and attributes in your database schema need to be to what the properties on the `Twitter` framework classes are (even though we did exactly that in the simple lecture demo).
6. You are very likely to want an attribute somewhere in your schema which is an integer.
7. All `NSManagedObject` instances know the `NSManagedObjectContext` they are part of (they have a `var` which returns it). You will likely find this useful. It will keep you from constantly having to go back to your `AppDelegate` to get the managed object context.
8. You'll recall from lecture that to-many relationships in the database are `NSSet`s in your code. `NSSet` is not mutable (i.e. you can't add anything to or remove anything from one). If you want to add or remove something from a to-many relationship, use the `func`s that are created for you by Xcode when you choose Category/Extension in the inspector for an Entity.
9. Remember that Core Data automatically updates the inverse of any relationship. So if you add something to a to-many relationship `NSSet`, it will automatically make the other side the the relationship correct (even if that other side is also a to-many relationship).
10. You can distinguish one Tweet from another using the `identifier` field in `Twitter.Tweet`. In other words, that `identifier` is unique across all Tweets in the universe.
11. Don't forget to `save()` your managed object context(s).

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. Application Delegate
3. `NSManagedObjectContext`
4. `NSFetchedResultsController`
5. Detail Disclosure Segues
6. Database schema design

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 visually 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, long methods, etc.
- UI is a mess. Things should be lined up and appropriately spaced to "look nice."
- 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.
- Public and private API is not properly delineated.
- The boundaries of MVC are violated.
- The Swift API Design Guidelines were not properly adhered to.
- Main thread is blocked waiting for the network or loading things into Core Data.

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

There are lots of ideas below. We certainly don't expect that you'll do all of them (and some are more difficult than others). Read through them and pick whichever ones intrigue you the most.

1. Divide your mentions popularity table into two sections: Hashtags and Users. Again, with the right database schema, this can be implemented in very few (maybe two?) lines of code.
2. Loading up a lot of data by querying for an existing instance in the database, then inserting if not found over and over again, one by one (like we did in lecture), can be pretty poor performing. Enhance your application to make this more efficient by checking for the existence of a pile of things you want to be in the database in one query (then only creating the ones that don't exist). The predicate operator `IN` might be of value here.
3. You are not required to ever delete anything from your database, however, it only needs information about the most recent searches so, over time, there's wasted space in there. Have your application prune the database of objects that are no longer of interest (i.e. can't be accessed from the UI) to keep it a manageable size. It's up to you to decide when is a good time to do such pruning.