Assignment IV: Smashtag Mentions

Objective

In this assignment, you will enhance the Smashtag application that we built in class to give ready-access to hashtags, urls, images and users mentioned in a tweet.

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

- This is a completely new application, so you will not need anything (but the knowledge you gained) from your first three homework assignments.
- You will need a Twitter account.
- This set of Twitter utility classes is required (or at least, will be extremely useful!) for this assignment.

Required Tasks

1. Enhance Smashtag from lecture to highlight (in a different color for each) hashtags, urls and user screen names mentioned in the text of a Tweet (these are known as "mentions"). Note that mentions are already located for you in each Tweet by Twitter and show up as `[IndexedKeyword]`s in the `Tweet` class in the supplied Twitter code.
2. When a user clicks on a Tweet, segue to a new `UITableViewController` which has four sections showing the "mentions" in the Tweet: Images, URLs, Hashtags and Users. The first section displays (one per row) any images attached to the Tweet (found in the `media` variable in the `Tweet` class). The last three show the items described in Required Task 1 (again, one per row).
3. Each section in the mentions table view should have an appropriate header.
4. If a section has no items in it, there should be no header visible for that section.
5. If a user touches an entry for a hashtag or a user in the "mentions table view" that you created in Required Task 2 above, you should segue to show the results of searching Twitter for that hashtag or user. It should be searching for hashtags or users, not just searching for a string that is the name of the hashtag or user (e.g. search for "#stanford", not "stanford"). The view controller to which you segue must work identically to your main Tweet-viewing view controller (`TweetTableViewController`).
6. If the user clicks on a mentioned url in your newly created view controller, you should open up that url in Safari (see Hints below for how to do that).
7. If the user clicks on an image in your newly created view controller, segue to a new MVC which lets the user scroll around and zoom in on the image. When the image first appears in the MVC, it should display zoomed (in its normal aspect ratio) to show as much of the image as possible but with no "whitespace" around it.
8. Keep track of the most recent 100 Twitter searches the user has performed in your application. Add a `UITabBarController` to your application with a tab for searching (i.e. your main UI) and a second tab showing these most recent search terms in a table view (uniqued with most recent first). When a user clicks on a search term in the second tab, segue (stay in that same tab) to show the most recent Tweets matching that search term. Store these most recent searches permanently in `NSUserDefaults` so that your application doesn't forget them if it is restarted.
9. You must not block the main thread of your application with a network request at any time.
10. Your application must work properly in portrait or landscape on any iPhone (this is an iPhone-only application).

Hints

1. You will need to log in to Twitter in Settings on your device (or on the simulator) to make the provided `Twitter` classes work.
2. The Twitter classes provided are `Printable`, so you can print them out with `println`.
3. Don't be overwhelmed by all the code in the Twitter classes. The only method you'll ever need to call in the entire framework is `fetchTweets`. Otherwise you just need to access whatever properties you need in the `Tweet`, `MediaItem` and `User` data structures.
4. Most UIKit classes (like `UILabel` and `UIButton`) have a method `attributedText` which lets you set and get its text using an `NSAttributedString`.
5. Make sure you do not "break" the feature that currently exists in Smashtag whereby it shows Tweets using the preferred body font style (and thus the text in the Tweets can be made larger or smaller by the user in Settings).
6. To add a `UITableViewController` to your storyboard, just drag one out of the Object Palette and change its class to be a custom subclass of `UITableViewController` you create using New File.
7. Your new "mentions" (and images) MVC has different "kinds" of things in each section. While you might be tempted to deal with this with large `if-then` or `switch` statements in your `UITableViewDataSource` and navigation methods, a cleaner approach would be to create an internal data structure for your `UITableViewController` which encapsulates the data (both the similarities and differences) in the sections. For example, it'd be nice if `numberOfSectionsInTableView`, `numberOfRowsInSection` , and `titleForHeaderInSection` were all "one-liners".
8. In fact, in general, any method that has more than a dozen lines of code is probably going to be hard for readers of your code to understand (and might well betray a "less than optimal" architectural approach).
9. Don't forget about Swift features like `enum`. Use Swift to its fullest. Harken back to the data structure we created for the `CalculatorBrain`. It might provide some inspiration for this assignment too.
10. As always, give solid thought as to what the "public (i.e. non-`private`) API" of your new controller is. Make everything else `private`. Your public API is what says to the rest of your application "this is how you use this controller." No other part of your application should know anything about the internal workings of your controller. And your controller should always "do the right thing (i.e. do what it was created to do)" when some other part of your application uses the controller by calling its public API.
11. Ditto for any `UITableViewCell` subclass you create. Or any class you create for that matter!

12. Also think about the titles of your MVCs (i.e. what appears in the navigation bar of a navigation controller when that MVC is being shown).
13. If you are going to be indexing into an `NSAttributedString`, you will want to use the `nsrange` property of `IndexedKeyword`, not the `range` property (since `NSAttributedString` indexes into an underlying `NSString`, not an underlying `String`).
14. If you have an `NSURL` named `url`, you can open it in Safari like this: `UIApplication.sharedApplication().openURL(url)`.
15. When you click on a user or hashtag in your mentions MVC, you can segue to the "list of Tweets" table view controller using either a normal "Show" segue or using an "Unwind" segue. It's up to you which of those you think results in a better user-interface.
16. You will almost certainly need two different `UITableViewCell` prototypes in your storyboard. Give them different identifiers and dequeue an appropriate one in `cellForRowAtIndexPath`.
17. Your new view controller's row heights don't need to be "estimated" like the row heights of the "list of Tweets" controller because you have very few rows and performance is not a consideration. Thus you will likely want to implement the `UITableViewDelegate` method `heightForRowAtIndexPath`.
18. For your rows that contain an image, you'll have to figure out an appropriate height. For the other rows in your table, you can just let them automatically figure their own height (using autolayout) by returning `UITableViewAutomaticDimension` from `heightForRowAtIndexPath`.
19. You can figure out the aspect ratio of an image in a `Tweet` without having to actually fetch the actual image from its url. See the `MediaItem` class in the Twitter classes provided.
20. A cool feature of your application is (should be!) that if the user wants to zoom in on a Tweet's image a bit without clicking on it to segue to the detailed image viewing MVC, the user can simply rotate the device to landscape. If you implement things properly, you'll get this feature "for free" (i.e. no code required).
21. For the required task where the user can click on an image to start panning and zooming on it in a new MVC, you can mostly reuse code from Cassini. However, you'll have to add the autozooming-to-fit capability to the `ImageViewController`.
22. It would be cool to make that autozooming-to-fit behavior continue to happen whenever the MVC's view's geometry changes until the user explicitly zooms with a gesture (there is a delegate method to find out when that occurs). That way it'll autozoom-to-fit as the user rotates their device.
23. It's probably a good idea to have a single, global "truth" for the most recent search terms and, since you have to store them in `NSUserDefaults` anyway, why not make

`NSUserDefaults` be that truth? You might want to wrap a little `class` or `struct` around your storing and recalling from `NSUserDefaults` that you can use throughout your application.

24. `NSData(dataWithContentsOfURL:)` **blocks** the thread it is called from when invoked with a network url. Thus you cannot call it from the main thread.
25. You cannot make any calls into UIKit from any thread other than the main thread. Be careful not to "accidentally" do this by calling some method which subsequently calls a method in UIKit. If you call a method from UIKit (directly or indirectly) off the main thread, your UI will fail in unpredictable ways.
26. The `fetchTweets` method executes its handler **off** the main thread.
27. Remember that the cells of a `UITableView` are only created for **visible** cells and they are **reused** as data comes on screen and goes off screen.
28. If you are fetching in a thread other than the main thread and then get the result and then want to ask the main queue to do something with that result, you'd better be sure nothing has "changed" while the network call was going on.

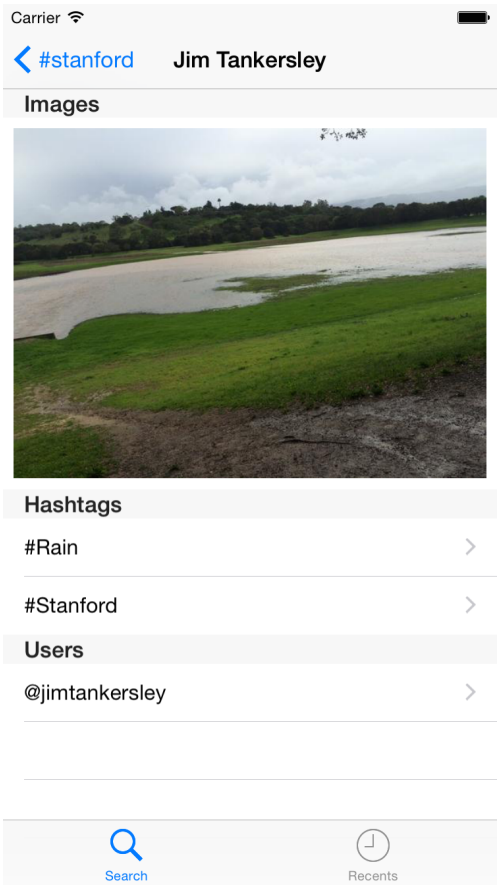
Things to Learn

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

1. `NSAttributedString`
2. `UITableView`
3. `UITableViewController`
4. `UITableViewCell`
5. `UIRefreshControl`
6. `UIActivityIndicatorView`
7. `UITabBarController`
8. Multithreading
9. Data structure design
10. `NSUserDefaults`
11. `UIScrollView`
12. `UIImageView`

Screen Shots

We are always hesitant to include screen shots because we don't want to restrict your creativity. These screen shots are NOT Required Tasks. They are just intended to give you an idea if you are having trouble visualizing the Required Tasks. The colors below were chosen completely at random. You should choose colors you think look good in your UI.



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.

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. In the Users section of your new `UITableViewController`, list not only users *mentioned* in the Tweet, but also the user who *posted* the Tweet in the first place.
2. When you click on a user in the Users section, search not only for Tweets that mention that user, but **also** for Tweets which were **posted** by that user.
3. If you segue using Show (rather than Unwind), add some UI which will Unwind all the way back to the `rootViewController` of the `UINavigationController`. Even if you use Unwind (rather than Show), then if do the Collection View extra credit below using a Show segue, you might want the "unwind to root" behavior in scenes you segue to via the Collection View.
4. Instead of opening urls in Safari, display them in your application by segueing to a controller with a `UIWebView`. You'll have to provide at least a little bit of "browser control" UI to go along with it (e.g. a "back button").
5. Make the "most recent searches" table be editable (i.e. let the user swipe left to delete the ones they don't like).
6. Add some UI which displays a new view controller showing a `UICollectionView` of the first image (or all the images if you want) in all the Tweets that match the search. When a user clicks on an image in this `UICollectionView`, segue to showing them the Tweet.

Extra Credit Hints

1. If you have built a good internal data structure for your section data, hopefully this is just a matter of enhancing an `init()` method for that internal data structure.
2. You will need to familiarize yourself with the "operators" in Twitter search queries.
3. You might find yourself in a situation where sometimes you want the "unwind to root" button to appear and sometimes you don't (for example, if you're already at the root, you clearly don't want it). You can try to manage this in code (with if then statements and showing or hiding the "unwind to root" button), or you could use polymorphism and have your root scene **not** have an "unwind to root" button and have its Controller implement the "unwind to root" `@IBAction` method that other scenes' "unwind to root" button would unwind to (it could do this by implementing nothing but this "unwind to root" method in a *subclass* of the non-root version of the MVC). Other scenes in the storyboard would have the button but wouldn't implement that method (because they use the superclass as their scene's class, not the subclass). As part of this, you might want to copy/paste entire scenes (and then change only their class). Then you can do this whole thing with unwinds only. Just food for thought. If this is confusing you, feel free to ignore this Hint and do it another way.
4. Check out the documentation for `UIWebView`.
5. When you create a `UITableViewController` subclass, the template will actually include some methods to help with this.
6. Here are some things to consider …
   - 6.a. The template you get when you create a subclass of `UICollectionViewController` has a call to `registerClass` in `viewDidLoad`. DELETE THIS LINE OF CODE. You will be setting the class of your `UICollectionViewCell`s in the storyboard instead. If you do not delete this call to `registerClass`, it will override anything you do in the storyboard since `viewDidLoad` gets called after the storyboard is done loading.
   - 6.b. Because you will obviously be downloading all those images off the main thread, scrolling around should be snappy, but, frankly, if you re-download them over and over as the user scrolls around, you'll get a lot of blank spaces that fill in over time and won't really look that great. So cache the images. Check out the class `NSCache`. It is like an `NSDictionary` (`objectForKey` and `setObject:forKey`), but adds the concept of a "cost" of something being in the cache via `setObject:forKey:cost:`. The "cost" of an image could be its size in kb, for example. The `NSCache` will throw things out of the cache any time it wants, so you will always just lookup the `NSURL` you want (to find the associated `UIImage`), use it if you find it, or just download it again if you don't. You'll want your cache associated with your `UICollectionViewController` subclass

(so that it will be shared by all the cells and so that it will go away when the controller goes away). You'll have to figure out the right way to make the cache available to the cells.

6.c. The big difference between a `UITableView` and a `UICollectionView` is that a table view is always laid out in exactly the same way (i.e. rows in a single column). A collection view has a `UICollectionViewLayout` property which determines how its cells are layout (and is thus massively flexible). `UICollectionViews` by default use a `UICollectionViewFlowLayout` to lay out its cells kind of like the characters in "justified text" are laid out. That should suit your purposes here just fine! Things like the size of a cell is determined by the `delegate` in both table views and collection views, but in collection views, the `delegate` responds to a protocol that is specific to its layout engine. For a `FlowLayout`, the protocol is called `UICollectionViewDelegateFlowLayout`. So if you want to control the size of cells, for example, you'd implement `collectionView:layout:sizeForItemAtIndexPath:` in your `UICollectionViewController` subclass.

6.d. You can take the easy way out and pick a predetermined size for the cells in the `UICollectionView` or, perhaps better, pick a predetermined "area" (i.e. width x height) for each one (but maintain each image's aspect ratio).

6.e. It would be cool to have "pinching" on the `UICollectionView` make the cell's size get larger and smaller (i.e. showing more or fewer images). Pinching should be trivial to implement if you take the approach above to size your cells (pinching would just scale the area up and down).

6.f. To show the tweet whose image gets clicked on, you can reuse your view controller that shows a list of tweets but you'll have to modify it to be able to be told to show a specific tweet(s) (in addition to still being able to search for tweets).