Programming Project #3
WSUV Twitter

CS 458

Due 11:59 pm, April 7, 2017

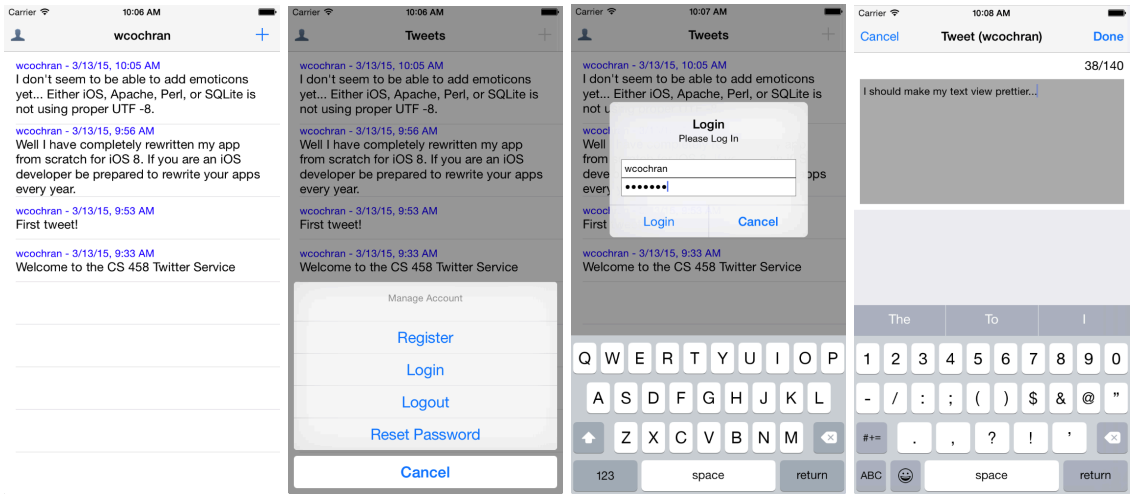

Figure 1: Screenshots: table of recent tweets, user management menu, logging in, and adding a tweet.

1 Introduction

For this project you will create a client iOS application for a Twitter-like service. User information and messages are stored on a server and are accessible via a web-based API using HTTP GET/POST methods. You will use the delightful *Alamofire* framework to communicate with the server. Your application will allow the user to...

- fetch and view the latest "tweets",
- authenticate (register, log-on, and log-off),
- post messages (authenticated users only), and
- delete messages that the user posted (bonus feature).

The project covers a variety of topics discussed in this course:

- UI controllers and views,
- keyboard input and user interaction,

- data persistence including using the iOS Keychain for secure storage of user information, and
- HTTP GET/POST communication with a RESTful API via Alamofire.

This document first explains how to store local tweet and user information in Section 2. A simple iPhone user interface described in Section 3. Section 4 explains how to use the Alamofire Library to communicate with the server; A variety of implementation details are given for fetching and adding tweets are given here. The backend API is specified in Section 6.

2 Model Objects

property	*description*
`tweet_id`	unique integer identifying tweet
`username`	string of user who posted tweet
`isdeleted`	true iff this tweet has been deleted
`tweet`	content of tweet (`NSString`)
`date`	time/date stamp of tweet (`NSDate`)

Table 1: Tweet object fields.

The tweets are stored remotely on the server, but are cached in a local array. The application delegate is a convenient place to store the tweets since it persists for the lifetime of the app, is easily accessible by all view controllers, and already has the hooks in place for loading and storing the tweets in the app's sandbox when the app launches and enters the background state. The example method below (from my main table view controller) demonstrates the pattern for accessing the app delegate and the cached tweets:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    return appDelegate.tweets.count
}
```

The tweets are returned from the server in an array of `NSDictionary`'s (which is automatically bridged to a Swift dictionary of type `[String : AnyObject]`) – one dictionary per tweet – as discussed in Section 4. For each dictionary I create my own `Tweet` object whose properties are listed in Table 1. The `date` property is created from the tweet's time stamp as follows:

```
let dateFormatter = DateFormatter()
dateFormatter.dateFormat = "yyyy-MM-dd HH:mm:ss"
dateFormatter.timeZone = NSTimeZone(abbreviation: "PST")
let date = dateFormatter.date(from:tweetDict["time_stamp"] as! String)
```

The tweets are sorted by ascending dates (*i.e.,* newest tweets are at the front of the array). My `Tweet` objects conform to the `NSCoding` protocol so they can be easily archived in the app's sandbox as described in class.

The only other model data is the current *username, password* and `session_token`; If no user is "logged on" these can all be `nil`. A SHA-1 hash of the password is stored on the server and, for security purposes, should be stored in the *Keychain* along with the `session_token`. Instead of sending the password to the server each time a tweet is added or deleted, a `session_token` hash is created by the server and is used during communication (see Section 6). This user information should be persistently stored along with the tweets.

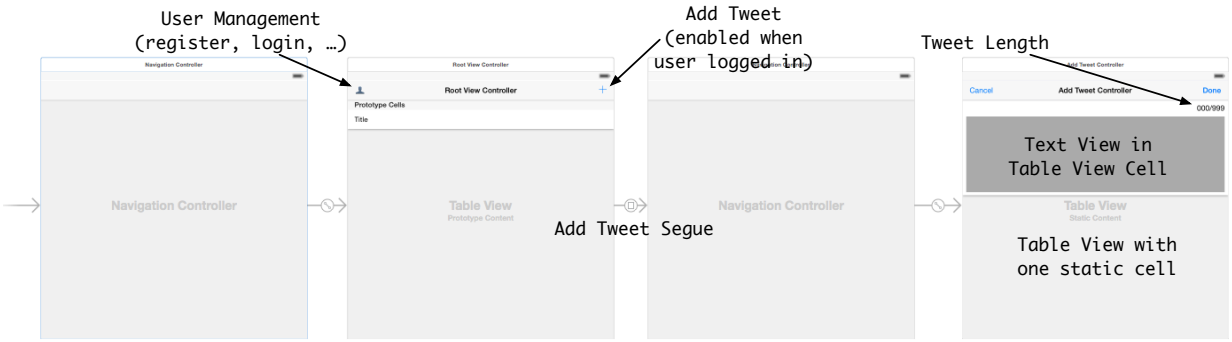

Figure 2: App storyboard consists of two content view controllers all embedded in navigation controllers.

3 User Interface

Figure 2 shows a storyboard containing two content view controllers each embedded in a navigation controller; Here the navigation controllers merely provide a navigation bar which is a convenient place to put titles and buttons – no controllers are ever "pushed onto" a navigation stack since they are all presented modally. These controllers define the primary interfaces for viewing tweets and adding a tweet. We can use `UIAlertControllers` for the user management menu and login view shown in the center of Figure 1 and described further in Section 3.2.

3.1 Viewing Tweets

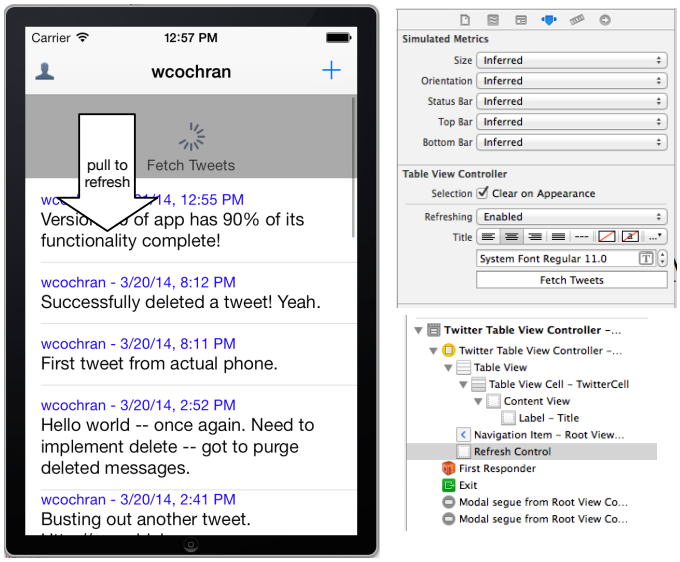

Figure 3: We use the "pull-to-refresh" gesture to fetch and reload tweets. The table view controller is configured to contain a refresh-control that we can connect outlets and actions to.

The primary controller is a table view controller that displays each tweet with the latest tweet near the top as shown on the left of Figure 1. The user can request that the latest tweets be fetched from the server by performing a "pull to refresh" gesture as illustrated in Figure 3. Adding a refresh control can be done using the IB inspector tool when configuring a table view controller as show on the right of the figure; This

```
@IBAction func refreshTweets(_ sender: AnyObject) {
   // If successfully fetched new tweets
   //    self.tableView.reloadData(); self.refreshControl?.endRefreshing();
   // If error
   //    display alert with message; self.refreshControl?.endRefreshing();
}

NotificationCenter.default.addObserver(
            forName: kAddTweetNotification,
            object: nil,
            queue: nil) { (note : Notification) -> Void in
                if !self.refreshControl!.isRefreshing {
                    self.refreshControl!.beginRefreshing()
                    self.refreshTweets(self)
                }
        }
```

Figure 4: The action method is triggered when user performs a "pull-to-refresh" gesture, but we can also initiate the refresh control programatically (*e.g.,* when the tableview is notified that a tweet has been added).

actually places a refresh control object in the NIB file that can be connected to an outlet and action. The code listed in Figure 4 shows the action that is triggered when the user performs the refresh gesture; this code also shows how the control can be programmatically initiated (*e.g.,* after the user adds a tweet).

3.1.1 Dynamically Sized Table View Cells

Since the length of each tweets varies between 1 and 140 characters, we display the tweets in table view cells with varying heights. Fortunately this is really easy now in iOS 9. I am just using a stock `UITableViewCell` with a *basic* style which uses a `UILabel` to display its content. If a `UILabel` needs to display text using more than one line we set its `numberOfLines` property to 0. Instead of setting the `UILabel`'s `text` property with a `NSString` we set its `attributedText` property with a handcrafted `NSAttributedString`:

```
cell.textLabel?.numberOfLines = 0 // multi-line label
cell.textLabel?.attributedText = attributedStringForTweet(tweet)
```

The code in Figure 5 crafts a string with the desired layout (colors, fonts, word-wrapping) and caches it in a dictionary so we do need need to reconstruct it the next time we need it. Then all we need to do is implement the following data source methods for our table view and the cell height will be automagically computed:

```
override func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
   return UITableViewAutomaticDimension
}

override func tableView(_ tableView: UITableView,
                        estimatedHeightForRowAt indexPath: IndexPath) -> CGFloat {
   return UITableViewAutomaticDimension
}
```

3.2 Modal dialogs with `UIAlertController`

The user management menu is presented as an "action sheet" (see the second image in Figure 1) using an `UIAlertController` as described earlier in the course. We can also use an `UIAlertController` to present an

```
lazy var tweetDateFormatter : DateFormatter = {
    let dateFormatter = DateFormatter()
    dateFormatter.dateStyle = .short
    dateFormatter.timeStyle = .short
    return dateFormatter
}()

let tweetTitleAttributes = [
    NSFontAttributeName : UIFont.preferredFont(forTextStyle: UIFontTextStyle.headline),
    NSForegroundColorAttributeName : UIColor.purple
]

lazy var tweetBodyAttributes : [String : AnyObject] = {
    let textStyle = NSParagraphStyle.default.mutableCopy() as! NSMutableParagraphStyle
    textStyle.lineBreakMode = .byWordWrapping
    textStyle.alignment = .left
    let bodyAttributes = [
        NSFontAttributeName : UIFont.preferredFont(forTextStyle: UIFontTextStyle.body),
        NSForegroundColorAttributeName : UIColor.black,
        NSParagraphStyleAttributeName : textStyle
    ]
    return bodyAttributes
}()

var tweetAttributedStringMap : [Tweet : NSAttributedString] = [:]

func attributedStringForTweet(_ tweet : Tweet) -> NSAttributedString {
    let attributedString = tweetAttributedStringMap[tweet]
    if let string = attributedString { // already stored?
        return string
    }

    let dateString = tweetDateFormatter.string(from: tweet.date as Date)
    let title = String(format: "%@ - %@\n", tweet.username, dateString)

    let tweetAttributedString = NSMutableAttributedString(string: title, attributes: tweetTitleAttributes)
    let bodyAttributedString = NSAttributedString(string: tweet.tweet, attributes: tweetBodyAttributes)

    tweetAttributedString.append(bodyAttributedString)
    tweetAttributedStringMap[tweet] = tweetAttributedString
    return tweetAttributedString
}
```

Figure 5: Constructing an `NSAttribuedString` for displaying a tweet in a table view cell. To avoid constructing this formatted string more than once, we cache it in the `tweet` object.

"alert" that is populated with textfields as illustrated in the third image in Figure 1. A "login" and "cancel" button are added to the alert as well as two textfield as listed in Figure 6. The actual authentication process is described in Section 4.4.

```
let alertController = UIAlertController(title: "Login", message: "Please Log in", preferredStyle: .alert)
alertController.addAction(UIAlertAction(title: "Login", style: .Default, handler: { _ in
     let usernameTextField = alertController.textFields![0]
     let passwordTextField = alertController.textFields![1]
     // ... check for empty textfields
     self.loginUser(usernameTextField.text!, password: passwordTextField.text!)
}))
alertController.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))
alertController.addTextField { (textField : UITextField) -> Void in
    textField.placeholder = "Username"
}
alertController.addTextField { (textField : UITextField) -> Void in
    textField.isSecureTextEntry = true
    textField.placeholder = "Password"
}
self.present(alertController, animated: true, completion: nil)
```

Figure 6: Using an Alert View for login.

4 Network Communication

Communication with the server requires creating HTTP connections which can be slow. Normally any time consuming processing needs to be performed on a background thread since blocking the main event-loop thread is never an option. Fortunately there are several frameworks that handle HTTP background network communication. `NSURLConnection` provides a mechanism for creating an HTTP connection for which you provide a delegate that can asynchronously receive data – the background network communication is handled for you. Since iOS 7, the URL loading system has been overhauled. The new `NSURLSession` class provides a richer set of features for HTTP processing. `Alamofire` is a third party Swift networking library built on top of Apple's API's that is even easier to use – here we describe how to use `Alamofire` for this project. `AFNetworking` is the analogous library for Objective-C.

4.1 Add Alamofire to your project

There are several ways to include Alomofire into your Xcode project. Perhaps the easiest way is to simply clone `Alamofire` from GitHub into a separate directory from your project and drag all the Swift files in the `Source` directory into your Xcode project (create a group named Alamofire to organize your files):

https://github.com/Alamofire/Alamofire

Make sure the the "copy items into destination group's folder" and the "create groups for any added folders" are both checked. Of course, if you do it this way you will need to repeat this process every time Alamofire is updated.

The documentation for Alamofire suggests some alternate methods. One popular method is to use *CocoaPods* which will require you to start by creating an Xcode workspace. I used a git *submodule* but it has been a little painful interacting with GitLab and it is the first time I have really used submodules so there have been some growth pains.

4.2 HTTP GET/POST Requests and JSON

You will use HTTP/GET requests with the server to fetch tweets and HTTP/POST requests for the remaining operations. Data is returned from the server in JSON format, but we will configure JSON responses to be automatically deserialized for us. Request parameters will be automatically URL encoded appropriately as well.

The server on the back-end is implemented with a set of Perl CGI scripts executed by the Apache web server over an HTTPS encrypted connection. The examples below use the following for the base URL string:

```
let kBaseURLString = "https://bend.encs.vancouver.wsu.edu/~wcochran/cgi-bin"
```

For development, please use

```
let kBaseURLString = "https://ezekiel.encs.vancouver.wsu.edu/~cs458/cgi-bin"
```

so you don't pollute the main tweet-feed. For all of our requests, we use the `Alamofire.request` method to initiate a HTTP POST or GET. We deserialize the response via the `responseJSON` method.

4.3 Fetching Tweets

In my implementation, the table view controller that displays the tweets is responsible for retrieving them when the user performs a "pull-to-refresh" gesture as explained in Section 3.1. The only parameter for the GET request is a time-stamp that specifies how far back in time to retrieve tweets. I determine the time-stamp for the latest tweet (which is always the first tweet since they are sorted by descending time-stamps) and use an `NSDateFormatter` to construct the appropriate string:

```
let dateFormatter = DateFormatter()
dateFormatter.dateFormat = "yyyy-MM-dd HH:mm:ss"
dateFormatter.timeZone = TimeZone(abbreviation: "PST")
let lastTweetDate = appDelegate.lastTweetDate()
let dateStr = dateFormatter.string(from: lastTweetDate as Date)
```

Figure 7 lists the method invocation that initiates the HTTP/GET request for the tweets. This method returns immediately after making the request, and the magic happens when a future response is received. The `Alamofire.request` method is used to initiate the HTTP/GET and the `.responseJSON` method will deserialize the response when it eventually arrives. We pass a function (a "closure") to `.responseJSON` that will be invoked on the *main thread* when a full response is received from the server (or a timeout occurs). Even though this function is invoked long after the enclosing method has returned, it will still have access to all the same local variables – this is the amazing thing about *closures*; If you are a JavaScript programmer this should be quite natural to you.

We determine if there was a success or failure by examining the `response.result` property. On `.Success,` the JSON encoded response will held in a value associate with `response.result.Success` enum type – this value is a dictionary that reflects the structure of the returned JSON described in Section 6.1. On `.Failure`, we determine what the error is and display an appropriate "alert" message to the user. In both cases, we send an `endRefreshing` method to the refresh control.

4.4 Authenticating

No authentication is necessary to view tweets, so the login view is only presented when the user requests to login (or register as a new user). The bar button item for adding a tweet is disabled if the user is not logged on. Figure 8 lists how to perform an HTTP/POST for a login (see Section 6.3).

4.5 Storing User Credentials in the Keychain

Both OS X and iOS use a *Keychain* service for sensitive information like passwords and session token's. Programmers can interface with the Keychain API using Apple's *Security Framework*, but it is somewhat complicated to learn how to use directly. Fortunately, there is a nice wrapper called *SSKeychain* on github that makes this easy. Instead of using a submodule I dragged the source code dragged this directly into my project and I had to change the following line in `SSKeychain.h`

```
#import <SSKeychain/SSKeychainQuery.h>
```

```
// format date string from latest stored tweet...
Alamofire.request(kBaseURLString + "/get-tweets.cgi", method: .get, parameters: ["date" : dateStr])
         .responseJSON {response in
             switch(response.result) {
             case .success(let JSON):
                let dict = JSON as! [String : AnyObject]
                let tweets = dict["tweets"] as! [[String : AnyObject]]
                // ... create a new Tweet object for each returned tweet dictionary
                // ... add new (sorted) tweets to appDelegate.tweets...
                self.tableView.reloadData() // force table-view to be updated
                self.refreshControl?.endRefreshing()
             case .failure(let error):
                let message : String
                if let httpStatusCode = response.response?.statusCode {
                    switch(httpStatusCode) {
                    case 500:
                       message = "Server error (my bad)"
                    // ...
                    }
                } else { // probably network or server timeout
                   message = error.localizedDescription
                }
                // ... display alert with message ..
                self.refreshControl?.endRefreshing()
             }
        }
```

Figure 7: Using `Alamofire.request` to fetch tweets asynchronously and `responseJSON` to deserialize the response. The closure passed to `responseJSON` will be called when the reponse is received over the network (or a timeout occurs). The `response.result` parameter encodes the eventual response.

to

```
#import "SSKeychainQuery.h"
```

If you install this as a framework, I suppose you do not need to do this.

Since *SSKeychain* is an Objective-C framework, Xcode will ask you to create a *bridging header* file so you access the code in Swift (see Figure 9). Insert the following lines into this newly created file:

```
#import "SSKeychain.h"
#import "SSKeychainQuery.h"
```

Storing and retrieving data is very simple; for example, storing a password is as simple as:

```
SSKeychain.setPassword(password, forService: kWazzuTwitterPassword, account: username)
```

I defined the following string in the app delegate to identify the "service" for storing the password for my app:

```
let kWazzuTwitterPassword = "WazzuTwitterPassword"  // KeyChain service
```

```
let urlString = kBaseURLString + "/login.cgi"
let parameters = [
   "username" : username,  // username and password
   "password" : password,  // obtained from user
   "action" : "login"
]
Alamofire.request(urlString, method: .post, parameters: parameters)
      .responseJSON(completionHandler: {response in
          switch(response.result) {
          case .success(let JSON):
             // save username
             // save password and session_token in keychain
             // enable "add tweet" button
             // change title of controller to show username, etc...
          case .failure(let error):
             // inform user of error
          }
      }
```

Figure 8: Logging on a user.

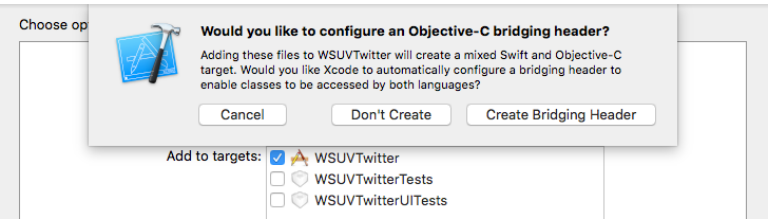

Figure 9: Xcode will ask to create a "bridging header" when importing Objective-C code into a Swift project.

4.6 Adding Tweets

The user adds a tweet by touching the "add" button on the top right of the navigation bar which should present the "Add Tweet" modal view controller as shown on the right of Figure 1. When the user finishes editing the tweet she touches the "done" button and the tweet is sent to the server via an HTTP/POST request with the required parameters (see Backend description in Section 6). If the POST was successful, then the controller should post a notification for which the main controller should be listening for and the table view controller refreshes the tweets with a HTTP/GET request as described earlier.

4.7 Deleting Tweets

As a bonus feature your app should allow the user to delete there own tweets. A delete can be initiated by by a finger swipe of a table-view cell which reveals a delete button as illustrated in Figure 10. Our API uses an HTTP/POST to delete a tweet (see API in Section 6). The tweet is not actually deleted, but is marked as deleted and its time-date stamp is updates so that other clients will be notified of the deleted tweet so they can purge it from their tableviews and caches. If successful, the deleted tweet can by simply removed from the tableview (no need to refresh the tweet).

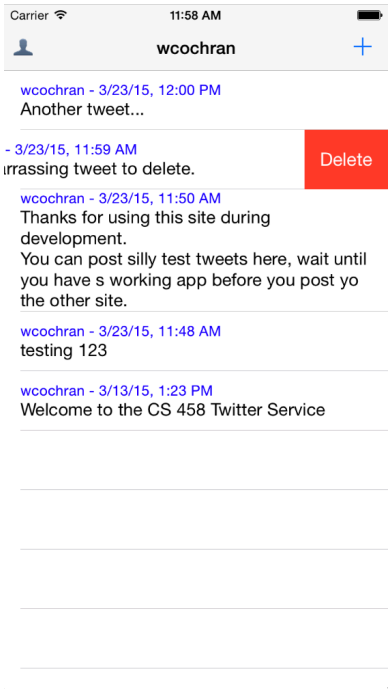

Figure 10: A selete initiated by a finger swipe of a table-view cell.

4.8 Network Activity Indicator

There is a framework associated with Alamofire that will automagically display the network activity indicator on the status bar when network requests are pending. You can find the framework at:

https://github.com/Alamofire/AlamofireNetworkActivityIndicator

I simple enable it when my app launches as follows

```
NetworkActivityIndicatorManager.sharedManager.isEnabled = true
```

and that it!

5 Submission

You are not required to mimic my interface or semantics exactly. One nice feature is to use dialog bubbles for displaying the tweets in the tableview. Submit your project by pushing to the ENCS gitlab repository manager. Make sure its (initially) private and add me as a member with Reporter access.

6 Backend API

`https://bend.encs.vancouver.wsu.edu/~wcochran/cgi-bin/`

6.1 Get Tweets

- HTTP/GET `get-tweets.cgi?date=<timestamp>`
- Example : `get-tweets.cgi?date=2013-03-14%2014:15:01`
- Description: Fetch all tweets with time stamps later than given date.
- Parameter `date` : time stamp of earliest tweet to fetch
- Returns (on success) JSON object

```
{"tweets" =
  [
    { "tweet_id" : 1,
      "username" : "wcochran",
      "time_stamp" : "2013-03-14 14:15:01",
      "isdeleted" : 0,
      "tweet" : "Welcome to the WSUV Twitter Service"},
     ...
   ]
 }
```

- Errors

  500 Internal Server Error : my bad;

  503 Database Unavailable : unable to connect to internal database.

6.2 Register New User

- HTTP/POST `register.cgi`
- Description: Register new user and logs user in.
- Parameters: `username, password`
- Returns (on success) JSON object with "session token"

```
{"session_token" : "765ADF654A64D566D6F6E66A"}
```

- Errors

  500 Internal Server Error : my bad;

  400 Bad Request : both username and password not provided

  409 Conflict : username already exists.

6.3 Login / Logout

- HTTP/POST `login.cgi`
- Description: Login / logout users.
- Parameters: `username, password, [action=login|logout]`
- Returns on successful login JSON object with "session token"

  ```
  {"session_token" : "765ADF654A64D566D6F6E66A"}
  ```

  On successful logout returns JSON object with session token 0

  ```
  {"session_token" : "0"}
  ```

- Errors

  500 Internal Server Error : my bad;

  400 Bad Request : both username and password not provided;

  401 Unauthorized : Unauthorized;

  404 Not Found : no such user.

6.4 Add Tweet

- HTTP/POST `add-tweet.cgi`
- Description: Add a tweet (up to 140 characters in length – actually database stores up to 200 characters).
- Parameters: `username, session_token, tweet`.
- Returns (on success) an echo back of the tweet:

  ```
  {"tweet" : "content of tweet"}
  ```

  Note that since neither the `tweet_id` nor actual time date stamp used is returned, the client is responsible for fetching tweet tweets (via `get-tweets.cgi`) to get the data for any new tweets (tweets later than its latest stored time stamp).

- Errors

  500 Internal Server Error : my bad;

  400 Bad Request : all parameters not provided;

  401 Unauthorized : Unauthorized;

  404 Not Found : no such user.

6.5 Delete Tweet

- HTTP/POST `del-tweet.cgi` (note not HTTP/DELETE)
- Description: Delete a tweet.
- Parameters: `username, session_token, tweet_id`.
- Returns (on success) JSON object:

  ```
  { "tweet_id" : 5342, "isdeleted" : 1, "tweet" : "[delete]" }
  ```

  The tweet is not actually deleted from the DB, but the content is replaced with "[delete]", the `isdeleted` field is set to true, and the `time_stamp` is updated so that that any clients will be notified that it has been deleted upon refresh.

- Errors

  500 Internal Server Error : my bad;

  400 Bad Request : all parameters not provided;

  401 Unauthorized : Unauthorized;

  403 Forbidded : not the user's tweet.

  404 Not Found : no such user or no such tweet.