FACEBOOK GRAPH API 26

SESHADRI MAHALINGAM
seshadri@berkeley.edu
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

August 05, 2010

1 Overview of the Facebook Graph API

The Facebook Graph API is a mechanism through which developers can request information from Facebook's servers. You can read and write data to Facebook using URL interactions. We've simplified the process in Scheme by providing you with methods to enable these interactions.

1.1 Really cool demo!

To give you a taste of what's to come, I'm going to post something to my Facebook feed.

```
(make-authorised-post-request (create-url-structure (list "me" "feed"))
                              (list (cons "message" "I <3 CS61A!")))
```

And voilà, I've posted to Facebook! What did I get back? As it turns out, everything in Facebook has an ID. You, me, your pet cat's fan page, every comment, every status update...

I can now query the returned ID number for information about the post.

```
(make-authorised-request (create-url-structure (list "<some number>"))
                         nil)
```

What else can I do with this? Let's check my post for "likes". ☺

```
(make-authorised-request (create-url-structure (list "<some number>" "likes"))
                         nil)
```

1.2 How does this work?

Here's a high-level overview of the process. If I type `https://graph.facebook.com/704520608` into a web browser, I can see my user account's public information. That's because `704520608` is my user ID. People and pages also have the option to choose a username for themselves, and we can use this username instead: `https://graph.facebook.com/704520608`

Every object in the social graph has a unique ID. We can retrieve information about an object by fetching `https://graph.facebook.com/ID`, no matter the type of the object.

Objects are also connected to each other via relationships, called connections. I can get a list of my friend connections using `https://graph.facebook.com/seshness/friends`.

1.3 Where's the documentation?

`http://developers.facebook.com/docs/api`

This website will be your bible during the course of the project. It contains a detailed Graph API Reference that expounds on each type of object in the social graph and lists the types of interactions it is capable of.

Each Scheme function you will need to use is commented in `facebook-hooks.scm`, with the inputs and outputs specified. If you feel that something is unclear or would like better explanations for ceratin functions, try the Project 4: Facebook Graph API thread on the Google group.

2 Our Scheme Way

2.1 Let's get some info

At the heart of our "hooks" into Facebook's API is a method `(make-request request options-alist)` that takes in a request and an association list of options and returns a familiar deep list/pair structure with the information we've requested. A typical Facebook Graph API URL has the format:

`https://graph.facebook.com/331218348435/attending?metadata=1&access_token=...`

where we're calling `331218348435/attending` the **request** and `metadata` & `access_token` the **options**.

We can use the `make-request` procedure in conjunction with `create-url-structure` to take care of the details involved in setting up this kind of URL and retrieving the relevant data with the following call:

```
(make-request (create-url-structure (list "331218348435" "attending"))
              (list (cons "metadata" "1") (cons "access_token" "...")))
```

Notice how we're double-quoting our arguments here. We're interacting with Scheme at a deeper level than we normally do, and hence mere words aren't enough. We need strings. Strings start and end with a double quote character and are considered atoms in STk. One important thing to note about strings is that two strings are never `eq?` to each other, but they can be `equal?`.

Certain options in a URL can have multiple values. The `fields` option, for example, returns specific properties of a requested object. A request like `https://graph.facebook.com/seshness?fields=id,name,gender` will retrieve only my Facebook ID, name and gender. We can produce this kind of request too by providing a *list of values* instead of a single value for an option in our association list:

```
(make-request (create-url-structure (list "seshness"))
              (list (cons "fields" (list "id" "name" "gender"))))
```

2.2 Post requests

Using `make-request`, we can retrieve information from Facebook. What about posting instead?

There is a `make-post-request` procedure that functions as one might expect. It takes a request, similar to that previously described, and an association list of arguments. For example, according to `http://developers.facebook.com/docs/reference/api/post` I can publish a wall post by adding the argument "`message`" with a value of "`om nom nom`" to a request of "`PROFILE_ID/feed`". The latter can be constructed easily using `create-url-structure`:

```
(create-url-structure (list "seshness" "feed")) => "seshness/feed"
```

The former is simply a list of pairs. Combining these inputs in make-post-procedure gives us:

```
(make-post-request (create-url-structure (list "seshness" "feed"))
                   (list (cons "message" "om nom nom")))
```

3 Authentication

So far we've been glossing over a pretty big detail - security! What prevents me, or my Facebook Graph enabled application, from posting to your wall? This is a recurring problem amongst "Web 2.0 apps", and many websites resolve this with an *access token* scheme.

3.1 The High-Level Idea

Facebook does not trust developers. At least not with usernames and passwords. No Facebook application should require a user to hand out his/her username and password. Instead, an application asks for permission to do specific things by going to a certain URL. A user would then log in and click the "Allow" button to allow that application to do its specific task(s). By clicking "Allow", the user acknowledges that they trust the application, and the application can now receive an *access token*. Adding the access token as an option to a request will authorize the request on behalf of the user, and the application will gain the perspective of the user.

For example, suppose I were coding an application to check my news feed for posts from friends with the words "om nom nom" and "Like" these posts automatically. I would have to ask my user for the "read_stream" and "publish_stream" permissions. They would then be redirected to Facebook, and see a fancy popup asking them if they want to allow or deny these permissions. When they click allow, the application will be granted an access token and will get access, using this access token, to the news feed and the ability to wall post/comment/like on posts. on behalf of the user.

These access tokens are valid for a limited amount of time, and normally will expire if the user logs out or at the end of the allocated amount of time, whichever comes first.

3.2 Registering your application

There's an equation for this ☺:

```
Access Token = User Login + Permissions + Application ID
```

Remember that everything in the Graph API has an ID, including your application! You can get an Application ID by registering your application at http://www.facebook.com/developers/ and clicking the "+ Set up New Application" button on the right hand side of the page. Give your application a name, and you will soon be able to view the application's administration page. We're only interested in the application ID from this page, but feel free to explore some of the settings.

This application ID allows Facebook to uniquely identify any single application, and remember an application's permissions automatically. We're now ready to get an access token and make some authorized requests!

3.3 Authenticating in Scheme

The first procedure to familiarize yourself with is `(put-application-id app-id)`. Like the name suggests, it takes an application ID as a string, and stores it in Scheme's default table (using put). If you want to retrieve it, you could use the `(get-application-id)` procedure.

Once you've stored your application ID in Scheme, you can call (launch-auth-page). This procedure accepts any number of permissions as arguments, so if we were coding the example application above, you would want to make the following call:

```
(launch-auth-page "read_stream" "publish_stream")
```

The launch-auth-page procedure opens a web browser and directs the user to Facebook. After logging in, they are presented with the Allow/Don't Allow prompt. Click allow, and copy the address of the page with the word "Success" on it. This web address contains the access token that we need to give our application using the put-access-token method:

```
(put-access-token "http://www.facebook.com/connect/login_success.html
                   #access_token=some_really_long_gibberish&expires_in=5118")
```

We can retrieve the stored access token by calling `(get-access-token)`. That's it - we've now successfully authenticated our application!

3.4 Making authorized requests

We're authenticated - great! Now how do we use our access token?

Every authorized request we make needs to have the access token added to the list of options (or arguments in the case of post requests). For example, to see a list of my friends now that I've authorized the application, I would call:

```
(make-request (create-url-structure (list "me" "friends"))
              (list (cons "access_token" (get-access-token))))
```

For an application making a call with an access token, "me" in the request refers to the logged in user.

We expect that your project will consist of making many authorized requests like the one above, and strongly recommend the use of helper procedures.