GitHub API Login]

[Board and Card Game App Team]

[Wyatt Derk]

[2/15/19]

[How do I create a GitHub login via my app?]

With the Board and Card Game App approaching the end of its first sprint, research must be taken to in order to add a GitHub login. Below are the steps needed to add this feature to the site.

Registering the app:

The application must first be registered as an OAuth application. This done in the following link: https://github.com/settings/applications/new . Most of the form will be standard information. When you get to the Authorization callback URL, use the following format: https://homepageurl/login/callback.

In the above link, the homepageurl stands for your homepage url. After that, create a callbackpage inside a login directory. There are other ways to create this, but this may be the most simple.

Once you are through the form be sure to copy the apps `Client ID` and `Client Secret.`

Accepting User Authorization:

Here a Ruby server (using Sinatra) implements the web flow of the app.

Create the server file called server.rb and paste into it:

```
require 'sinatra'
require 'rest-client'
require 'json'

CLIENT_ID = ENV['GH_BASIC_CLIENT_ID']
CLIENT_SECRET = ENV['GH_BASIC_SECRET_ID']

get '/' do
  erb :index, :locals => {:client_id => CLIENT_ID}
end
```

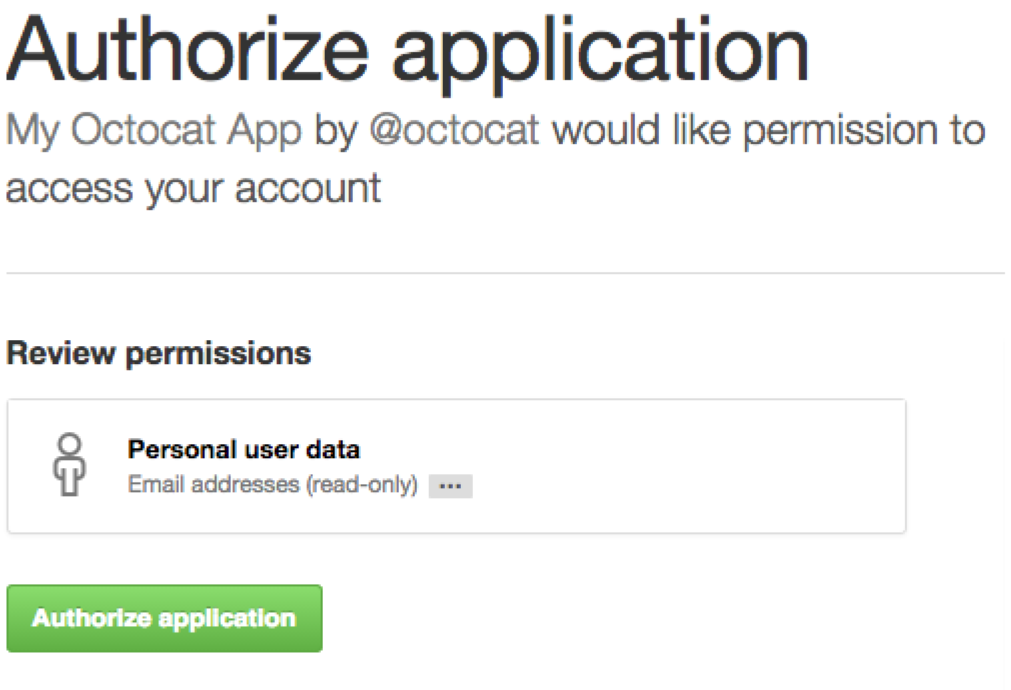
Here you can see the client ID and client secret are entered and stored as environment variables.

Next we look at this link in the example below:

```
<html>
  <head>
  </head>
  <body>
    <p>
      Well, hello there!
    </p>
    <p>
      We're going to now talk to the GitHub API. Ready?
      <a href="https://github.com/login/oauth/authorize?scope=user:email&client_id=<%=
client_id %>">Click here</a> to begin!</a>
    </p>
    <p>
      If that link doesn't work, remember to provide your own <a href="/apps/building-oauth-
apps/authorizing-oauth-apps/">Client ID</a>!
    </p>
  </body>
</html>
```

Above is a sample page created from a tutorial referenced below. We simply need to pay attention to the URL's. Notice in the first URL, a scope query parameter is defined to request reading private email addresses. Scopes, more specifically are what let developers specify what access they need. Other available scopes can be found here: https://developer.github.com/apps/building-oauth-apps/understanding-scopes-for-oauth-apps/

That first link will direct the user to GitHub with page similar to this:


Authorize application

My Octocat App by @octocat would like permission to access your account

Review permissions

Personal user data

Email addresses (read-only)

The “Authorize application” button will then send the user to the Callback URL. As of right now a 404 error will be spit out, because nothing more is defined.

Providing the Callback:

In server.rb, we add a route to specify what the callback is supposed to do. Here we add the following:

```
get '/callback' do
  # get temporary GitHub code...
  session_code = request.env['rack.request.query_hash']['code']

  # ... and POST it back to GitHub
  result = RestClient.post('https://github.com/login/oauth/access_token',
                          {:client_id => CLIENT_ID,
                           :client_secret => CLIENT_SECRET,
                           :code => session_code},
                           :accept => :json)

  # extract the token and granted scopes
  access_token = JSON.parse(result)['access_token']
end
```

Once an app is successfully authenticated, GitHub provides a tempory code value. This code will need to `POST` back to GitHub in exchange for an `access_token.` The tutorial shown, does so through the rest-client.

Below is a method for checking that the scopes were granted for the token by the user.

```
get '/callback' do
  # ...
  # Get the access_token using the code sample above
  # ...

  # check if we were granted user:email scope
  scopes = JSON.parse(result)['scope'].split(',')
  has_user_email_scope = scopes.include? 'user:email'
end
```

Here the scopes.include is used to check if user:email was granted. If we were asking for other scopes, those would be checked for as well.

Authenticating Requests:

Here, requests are authenticated as logged in users:

```
# fetch user information
auth_result = JSON.parse(RestClient.get('https://api.github.com/user',
                                        {:params => {:access_token => access_token}}))

# if the user authorized it, fetch private emails
if has_user_email_scope
  auth_result['private_emails'] =
    JSON.parse(RestClient.get('https://api.github.com/user/emails',
                              {:params => {:access_token => access_token}}))
end

erb :basic, :locals => auth_result
```

Below, the results are used to display various messages on the page.

```
<p>Hello, <%= login %>!</p>
<p>
```

```
  <% if !email.nil? && !email.empty? %> It looks like your public email address is <%= email
%>.
  <% else %> It looks like you don't have a public email. That's cool.
  <% end %>
</p>
<p>
  <% if defined? private_emails %>
  With your permission, we were also able to dig up your private email addresses:
  <%= private_emails.map{ |private_email_address| private_email_address["email"] }.join(', ')
%>
  <% else %>
  Also, you're a bit secretive about your private email addresses.
  <% end %>
</p>
```

Creating Persistent Authentication:

Lastly, we need to create a more persistent means of authentication, or else users need to login every time they access a web page. This is done with sessions.

We also need to handle cases when the user updates the scopes after we checked them, or revokes the token. We use the `rescue` block and check that the first API call succeeded, which verifies that the token is still valid. After that, we'll check the `X-OAuth-Scopes` response header to verify that the user hasn't revoked the `user:email` scope.

Below we have the updated server file with these commands.

```
require 'sinatra'
require 'rest_client'
require 'json'

# !!! DO NOT EVER USE HARD-CODED VALUES IN A REAL APP !!!
# Instead, set and test environment variables, like below
# if ENV['GITHUB_CLIENT_ID'] && ENV['GITHUB_CLIENT_SECRET']
#  CLIENT_ID        = ENV['GITHUB_CLIENT_ID']
#  CLIENT_SECRET    = ENV['GITHUB_CLIENT_SECRET']
# end

CLIENT_ID = ENV['GH_BASIC_CLIENT_ID']
CLIENT_SECRET = ENV['GH_BASIC_SECRET_ID']
```

```ruby
use Rack::Session::Pool, :cookie_only => false

def authenticated?
  session[:access_token]
end

def authenticate!
  erb :index, :locals => {:client_id => CLIENT_ID}
end

get '/' do
  if !authenticated?
    authenticate!
  else
    access_token = session[:access_token]
    scopes = []

    begin
      auth_result = RestClient.get('https://api.github.com/user',
                                   {:params => {:access_token => access_token},
                                    :accept => :json})
    rescue => e
      # request didn't succeed because the token was revoked so we
      # invalidate the token stored in the session and render the
      # index page so that the user can start the OAuth flow again

      session[:access_token] = nil
      return authenticate!
    end

    # the request succeeded, so we check the list of current scopes
    if auth_result.headers.include? :x_oauth_scopes
      scopes = auth_result.headers[:x_oauth_scopes].split(', ')
    end

    auth_result = JSON.parse(auth_result)
    if scopes.include? 'user:email'
      auth_result['private_emails'] =
        JSON.parse(RestClient.get('https://api.github.com/user/emails',
                       {:params => {:access_token => access_token},
                        :accept => :json}))
    end

    erb :advanced, :locals => auth_result
  end
end

get '/callback' do
  session_code = request.env['rack.request.query_hash']['code']

  result = RestClient.post('https://github.com/login/oauth/access_token',
                          {:client_id => CLIENT_ID,
                           :client_secret => CLIENT_SECRET,
                           :code => session_code},
                           :accept => :json)

  session[:access_token] = JSON.parse(result)['access_token']

  redirect '/'
end
```

Next, create a file in views called advanced.erb, and paste this markup into it:

```
<html>
  <head>
  </head>
  <body>
    <p>Well, well, well, <%= login %>!</p>
    <p>
      <% if !email.empty? %> It looks like your public email address is <%= email %>.
      <% else %> It looks like you don't have a public email. That's cool.
      <% end %>
    </p>
    <p>
```

```
      <% if defined? private_emails %>
      With your permission, we were also able to dig up your private email addresses:
      <%= private_emails.map{ |private_email_address| private_email_address["email"] }.join(',
') %>
      <% else %>
      Also, you're a bit secretive about your private email addresses.
      <% end %>
    </p>
  </body>
</html>
```

From the command line, call `ruby advanced_server.rb`, which starts up your server on port 4567 -- the same port we used when we had a simple Sinatra app. When you navigate to `http://localhost:4567`, the app calls `authenticate!` which redirects you to `/callback`. `/callback` then sends us back to `/`, and since we've been authenticated, renders *advanced.erb*.

[https://developer.github.com/v3/guides/basics-of-authentication/] /*This is the link to the tutorial usd to create the document. Some of the tutorial involved creating a Sinatra server via Ruby.*/