G54CCS Labs, Exercise 7

This exercise demonstrates making use of a third-party service, in this case the Twitter search service. Specifically we will examine:

- Maintaining synchronization state;
- Encoding data for the web;
- Compliant fetching data from a third-party service; and
- GAE Cron jobs and Task queues.

This exercise builds particularly on the storage exercise (#4), and to some extent on the URL parameter exercise (#3). It is quite a bit more complex than those you have seen so far, so do not be surprised if you need to work through it fairly slowly and carefully.

Background

JSON Encoding

We will encode the data we fetch from Twitter as *JSON*. This is a relatively simple textual encoding of potentially binary data. This will all be handled by a Python library that comes with GAE — for more information see `http://json.org/`. The two methods of interest are `json.loads(s)` which decodes string `s` into a Python object; and `json.dumps(v)` which encodes value `v` as JSON string.

Extra Python Syntax

We will use two new pieces of Python syntax. The first is an abbreviation. Instead of writing

```
if condition:
    result = value
else:
    result = othervalue
```

we can write

```
result = value if condition else othervalue
```

The second is also an abbreviation, albeit a little more complex: called a *list comprehension*, it is a compact way of constructing one list from another. As some of the data we get back from Twitter arrives in lists, this will be quite convenient. Thus, instead of

```
result = []
for item in list:
    if condition: result.append(process(item))
```

we can write

```
result = [ process(item) for item in list if condition ]
```

For example,

```
result = [ x*x for x in range(10) ]
```

Synchronization State

Since things can go wrong when invoking a third-party service, for a whole range of reasons that are nothing to do with us (e.g., network errors), and since a user may try to fetch data from us while we're still fetching data from Twitter, we need to record the *synchronization state* of our program. We will use three states for this:

- *unsynchronized*, we are yet to start fetching data from Twitter;
- *inprogress*, we are in the middle of fetching data from Twitter; and
- *synchronized*, we have fetched all the data that Twitter offered us at this time.

Structure

The overall structure of the program is as usual: `urls.py` contains the mappings from URL to handler class; `views.py` contains the handler classes themselves; `models.py` contains the mapping from the application data model to the GAE storage service.

In outline, the app will invoke the Twitter search API URL to fetch tweets that contain a particular hashtag (`#cloud`). Twitter limit the number of responses that may be returned to a single request, instead returning a `next_page` field with the results. Thus we must repeatedly invoke the `next_page` URL we are given until there are no more results. As GAE limits the length of time you can

spend processing a single invocation of a URL (typically to 30s), each repeated invocation of the `next_page` URL is via the *Taskqueue*. We schedule periodic fetching of new matching tweets using the GAE *Cron* facility: this causes GAE to periodically invoke a specified URL — in our case, every 2 hours. The Cron configuration is stored in a new file, `cron.yaml`.

cron.yaml

This is quite simple: each stanza contains an (optional) human readable *description*, the *URL* to be invoked, and the *schedule* for the URL to be invoked:

```
cron:
- description: twitter poller
  url: /cron
  schedule: every 2 hours
```

urls.py

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

import views

urls = [
    (r'/$', views.Root),
    (r'/cron/?$', views.Cron),
    (r'/sync(?:/(?P<cmd>start|stop))?/?$', views.Sync),
    (r'/tweets/?$', views.Tweets),
    ]

application = webapp.WSGIApplication(urls, debug=True)
def main(): run_wsgi_app(application)
if __name__ == "__main__": main()
```

This should now be fairly familiar in structure. We have four handler classes (`views.Root`, `views.Cron`, `views.Sync`, `views.Tweets`), and the usual application boilerplate. The `/sync` URL has a second element, `/start` or `/stop`, indicating whether synchronization should start or stop.

models.py

We begin with some relatively uninteresting boilerplate — you should be able to work out what all of this does now!

```
import sys, logging, datetime, time
log = logging.info
err = logging.exception

from google.appengine.ext import db
```

Next we have a class we use to enumerate possible states of our program: unsynchronized, in progress, synchronized:

```
class SYNC_STATUS:
    unsynchronized = 'UNSYNCHRONIZED'
    inprogress = 'INPROGRESS'
    synchronized = 'SYNCHRONIZED'
```

Wrapping the states in a class like this makes it harder to introduce bugs by mis-spelling them!

Then we have the class that actually records which state we're in and, if synchronized, when we last became so. The class also contains two helper methods to encode the state itself: as a Python dictionary (`todict`) and as a JSON string (`tojson`):

```
class SyncStatus(db.Model):
    status = db.StringProperty(default=SYNC_STATUS.unsynchronized, required=True)
    last_sync = db.DateTimeProperty()

    def put(self):
        if self.status == SYNC_STATUS.synchronized:
            self.last_sync = datetime.datetime.now()
        super(SyncStatus, self).put()
```

Finally we have the class we use to represent an individual tweet, both in `raw` form and after being JSON encoded (as text):

```
class Tweet(db.Model):
    raw = db.TextProperty(required=True)
    txt = db.TextProperty(required=True)
```

views.py

First, the usual boilerplate, plus two constants and the `render()` function you saw in exercise #5. The first constant indicates the number of results we will request each time we query Twitter; the second is the query string we will search for.

```
import logging, urllib, urlparse, datetime, os
log = logging.info

from google.appengine.ext import webapp
from google.appengine.ext.webapp import template
from google.appengine.api import urlfetch, users
from google.appengine.api.labs import taskqueue
from django.utils import simplejson as json

import models

COUNT = 80
HASHTAG = '#cloud'

def render(page, context):
    return template.render(os.path.join("templates", page), context)
```

The first handler class is quite simple: when the client issues a `GET` to `/`, simply return a JSON encoded list of all the tweets we have collected so far, ordered by the key under which they've been stored:

```
class Root(webapp.RequestHandler):
    def get(self):
        tweets = [
            json.loads(t.raw) for t in models.Tweet.all().order('__key__') ]

        self.response.headers['Content-Type'] = 'application/json'
        self.response.out.write(json.dumps(tweets))
```

The next class handles invocations on `/cron`, expected to be due to the GAE Cron service. The Cron service issues a `GET` to the specified URL (in our case, `/cron`) on the specified schedule (in our case, every 2 hours). In this case, all that causes is an entry to be put into the default taskqueue which, when executed, will issue a `POST` to `/sync/start`, starting a synchronization with Twitter:

```
class Cron(webapp.RequestHandler):
    def get(self):
        taskqueue.add(url="/sync/start", method="POST")
```

The next class is more interesting. This is what is invoked when requests are issued to `/sync/start` or `/sync/stop`. First, if the request is a `GET` to any URL under `/sync`, then we return a page (`templates/sync.html`) that tells us the current status and last synchronized time, and gives us buttons to manually start and stop synchronization.

```
class Sync(webapp.RequestHandler):
    def get(self, cmd):
        ss = models.SyncStatus.get_by_key_name("twitter-status")
        context = { "now": datetime.datetime.now().isoformat(),
                    "state": ss,
                    }
        self.response.out.write(render("sync.html", context))
```

If the request is a `POST` then:

- if the command (`cmd`) is `start`, and we're not *already* synchronizing (`SYNC_STATUS.inprogress`), then enter a request into the taskqueue to start synchronization.
- if the command is `stop`, then if we're currently in the middle of synchronizing (`inprogress`) we mark our status as unsynchronized so that no further results will be requested from Twitter. Then, as there may already be requests in the taskqueue waiting to execute, we purge the taskqueue of any outstanding tasks.

Finally, we return the sync status page as before.

```
    def post(self, cmd):
        if cmd == "start":
            s = models.SyncStatus.get_or_insert("twitter-status")
            if s.status != models.SYNC_STATUS.inprogress:
                taskqueue.add(url="/tweets/", method="GET")
                s.status = models.SYNC_STATUS.inprogress

            s.put()

        elif cmd == "stop":
            s = models.SyncStatus.all().get()
            if s.status == models.SYNC_STATUS.inprogress:
                s.status = models.SYNC_STATUS.unsynchronized
                s.put()
            taskqueue.Queue("default").purge()

        context = { "now": datetime.datetime.now().isoformat(),
                    "state": s,
                    }
        self.response.out.write(render("sync.html", context))
```

The last handler class is the one that actually causes results to be fetched from Twitter. This has just one method handler, for `GET`. In outline what it does

is issue an appropriate URL fetch from the Twitter search service, process the results (either storing valid results or registering an error), and finally issue a follow-on request to get the next page of results if required.

Taking each segment in turn, first we construct the request to Twitter:

```
class Tweets(webapp.RequestHandler):
    def get(self):
        log("request:%s" % (self.request,))
        ps = { 'q': HASHTAG, 'rpp': COUNT, }
        max_id = self.request.GET.get("max_id")
        if max_id: ps['max_id'] = max_id

        page = self.request.GET.get("page")
        if page: ps['page'] = page

        url = "https://search.twitter.com/search.json?%s" % (
            urllib.urlencode(ps),)
        log("url:%s" % url)
```

Note that we have to use the `https` form of the URL to avoid the university's web proxy causing the development server to fail to contact Twitter. In a live server you would not need to do this, and could use the `http` form instead. .

Then we issue the request and decode the JSON-encoded results:

```
        res = urlfetch.fetch(url)
        log("res:%s\nhdr:%s" % (res.content, res.headers))
        js = json.loads(res.content)
```

Then we process the results; first, if the result was an error then check if it was because Twitter is limiting the rate at which we can use their search service. If so then reissue the request by adding it to the taskqueue to be executed as far in the future as Twitter have asked us to:

```
        if 'error' in js: ## retry after specified time
            retry = int(res.headers.get('retry-after', '300'))
            rurl = "%s?%s" % (self.request.path, self.request.query_string)
            log("retry: url:%s countdown:%d" % (rurl, retry+2,))
            taskqueue.add(url=rurl, countdown=retry+2, method="GET")
```

Otherwise we got a valid result! First, extract the tweets from the result, and store each one using the Twitter issued tweet id (`id_str`) as its key:

```
        else:
            if 'results' in js:
                for tw in js['results']:
                    t = models.Tweet.get_or_insert(
                        tw['id_str'], raw=json.dumps(tw), txt=tw['text'])
                    t.put()
```

Then check whether we need to fetch a further page of results, and if so, construct the correct (relative) URL and add a suitable request to the taskqueue:

```
            if 'next_page' in js:
                next_page = "%s%s" % (self.request.path_url, js['next_page'],)
                urlo = urlparse.urlsplit(next_page)
                next_page_rel = urlparse.urlunsplit(
                    ("", "", urlo.path, urlo.query, urlo.fragment))
                taskqueue.add(url=next_page_rel, method="GET")
```

If not, then we must have completed synchronization, so record that fact:

```
            else:
                s = models.SyncStatus.get_by_key_name("twitter-status")
                s.status = models.SYNC_STATUS.synchronized
                s.put()
```

Finally, return the result set — this will normally be ignored since the request will have issued from the taskqueue, but it's possible someone will have issued the request manually:

```
        self.response.headers['Content-Type'] = 'application/json'
        self.response.out.write(json.dumps(js, indent=2))
```

Page templates and CSS

templates/sync.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Twitter Sync Service: Control</title>
    <meta charset="utf-8" />
    <link href="/static/style.css" rel="stylesheet" type="text/css" />
  </head>
```

```
  <body>
    <p>Time now is {{ now }}.</p>
    <div id="syncstate">
      {% if state.status %}
      <p>Current synchronization state is {{ state.status }}</p>
      {% endif %}

      <p>
        {% if state.last_sync %}
        Last completed synchronization was at {{ state.last_sync }}
        {% else %}
        Never synchronized!
        {% endif %}
      </p>

    </div>

    <div class="button">
      <form action="/sync/start" method="post" accept-charset="utf-8">
        <input id="start_button" type="submit" value="start"/>
      </form>
    </div>

    <div class="button">
      <form action="/sync/stop" method="post" accept-charset="utf-8">
        <input id="stop_button" type="submit" value="stop"/>
      </form>
    </div>

  </body>
</html>
```

static/style.css

```
.button {
    float: left;
}

#mycounter {
    border: 1px solid red;
}

.counter {
    border-bottom: 1px dotted gray;
}
```

Exercises

Note that you may need to clear your development server's datastore between invocations!

Ex.1. Observe your application's behaviour in the log screen. Try loading the `/cron` page in your browser — what happens, and why?

Ex.2. Still observing the log screen, now try issuing a request to `/sync`, and clicking on the start/stop buttons you should see displayed.

Ex.3. Modify the application to search for tweets containing some other string.

Ex.4. Modify the application to retrieve results at a lower or higher rate. What effect does this have on your application's behaviour?