G54CCS Labs, Exercise 6

This exercise introduces the notion of a *ticket*, a server-generated token that uniquely identifies interaction with the service at some granularity, e.g., single transaction, single client. Specifically we will examine:

- Generating unique tickets;
- Utilising tickets; and
- Managing tickets.

This exercise builds particularly on the storage exercise (#4) and the URL parameters exercise (#3).

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.

In outline, the app will store and manipulate each counter under a unique ticket. You can think of each ticket as mapping to a user, although we do not store any user details with the ticket, and in practice you would not expose the user id in the URL as we will. By generating a ticket a client creates a 'slot' on the server under which it can carry out operations without risk of collision with any other client.

urls.py

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

import views

urls = [
    (r'/value/?', views.Value),
    (r'/value/(?P<ticket>[a-fA-F0-9]+)/?', views.Value),

    (r'/incr/?', views.Incr),
    (r'/incr/(?P<ticket>[a-fA-F0-9]+)/?', views.Incr),
    ]

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

This should now be fairly familiar in structure. As before, it simply defines two urls, `/value` and `/incr` with their associated handlers. However, this time it adds the possibility to explicitly specify the ticket — a string of hexadecimal digits, `a..f` or `A..F` or `0..9` — for which you wish to invoke either the `/value` or `/incr` operations. That is, the following are all valid URLs for this application:

- `http://localhost:8080/value`
- `http://localhost:8080/value/`
- `http://localhost:8080/value/abcdefg1234`
- `http://localhost:8080/incr`
- `http://localhost:8080/incr/`
- `http://localhost:8080/incr/1234235345632fcff`

models.py

```
from google.appengine.ext import db

class Ticket(db.Model):
    value = db.StringProperty()
    ctime = db.DateTimeProperty(auto_now_add=True)

    def put(self):
        n = db.GqlQuery("SELECT * FROM Ticket WHERE value=:v",
                        v=self.value).count()
        if n > 0:
            raise ValueError("collision! n=%d v=%s" % (n, self.value,))
        super(Ticket, self).put()

class Counter(db.Model):
    ctime = db.DateTimeProperty(auto_now_add=True)
    atime = db.DateTimeProperty(auto_now=True)
    value = db.IntegerProperty(default=0, required=True)
    ticket = db.ReferenceProperty(
        Ticket, collection_name="counters", required=True)
```

The first line is boilerplate, importing the relevant GAE library.

The `Ticket` class is new: it stores a `value` and the `ctime` (*creation time*) of a ticket. The `put` method is customised to ensure that when the ticket is saved, an exception is raised if it collides with any previously stored ticket. In this way we try to guarantee that we never store two tickets with identical values.

The `Counter` class is very similar to the class you used in exercise #4. The difference is that it adds a new property, `ticket` which is a *reference* to a previously stored ticket, i.e., it does not store the value of a ticket directly, but 'points to' an existing ticket. This is how we tie each counter to a particular ticket. Note the use of the `collection_name` field — this effectively creates a property called `counters` in each `Ticket` which will contain a list (of length one) of all the counters that refer to that ticket.

The remainder of the code is, as usual, in `views.py`.

views.py

```
import logging, datetime, hashlib, os

from google.appengine.ext import db
from google.appengine.ext import webapp
from google.appengine.ext.webapp import template

import models

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

def create_ticket():
    count = db.GqlQuery("SELECT * FROM Ticket").count()
    now = datetime.datetime.now().isoformat()
    return hashlib.sha1("%s:%s" % (count, now)).hexdigest()
```

The standard `views.py` boilerplate, with two additions: `render(page, context)` is a shortcut to render a `page` using a given `context` dictionary; and `create_ticket()` generates and returns a new ticket value.

The value generated by `create_ticket()` should be unique — it uses a standard function to hash the current time and the number of tickets created so far. There is a very — *very* — small chance that the computed hash value will collide for two different (`count`, `now`) pairs, but we can safely ignore that for now.

We continue:

```
class Value(webapp.RequestHandler):
    def get(self, ticket=None):
        logging.info("ticket:%s" % ticket)
        if ticket:
            ticket = models.Ticket.all().filter("value =", ticket).get()
        now = datetime.datetime.now().isoformat()
```

```
        counter = None
        counters = []
        if ticket: counter = models.Counter.all().filter("ticket =", ticket).get()
        else:
            counters = models.Counter.all().fetch(10)

        context = { 'now': now,
                    'ticket': ticket,
                    'counter': counter,
                    'counters': counters,
                    }
        if counter: counter.put()
        for c in counters: c.put()
        self.response.out.write(render("page.html", context))
```

The `Value` class is used to handle all requests to view currently held counter values. It looks a little complex but can be broken down into three sections, separated by blank lines.

1. In the first section we log any `ticket` value supplied by the user (defaulting to `None` if the user did not supply a ticket value), we lookup the actual `Ticket` object with that value, and we compute the current time.
2. In the second section we get the `Counter` objects we need to display — either the specific counter for the given ticket, or all the counters if no ticket was specified.
3. In the third and final section we construct the `context` dictionary, save any counters that we accessed so that their `atime` field is updated, and finally we render and return the page.

We continue:

```
class Incr(webapp.RequestHandler):
    def post(self, ticket=None):
        logging.info("ticket:%s ticket?%s" % (ticket, True if ticket else False))
        now = datetime.datetime.now().isoformat()

        if ticket:
            ticket = models.Ticket.all().filter("value =", ticket).get()
        if not ticket:
            ticket = models.Ticket(value=create_ticket())
            ticket.put()

        counter = models.Counter.all().filter("ticket =", ticket).get()
```

```
        if not counter: counter = models.Counter(ticket=ticket)
        else:
            counter.value += 1

        context = { 'now': now,
                    'ticket': ticket,
                    'counter': counter,
                    }
        counter.put()

        self.redirect("/value/%s" % ticket.value)
```

Finally, the `Incr` class handles all increment requests for tickets, including the case where no ticket has been specified and so one needs to be created. Again this looks a little complex but can be broken down into four sections:

1. In the first section we have the usual boilerplate to log any specified `ticket` and to compute the current time.
2. In the second section we manage tickets. First we see whether a ticket value was specified — if it was, then we retrieve the corresponding `Ticket` object. Then if either no ticket was specified *or* one was specified but no corresponding `Ticket` object exists, we create and then store a new `Ticket` object.
3. In the third section we manage counters. First we retrieve the `Counter` corresponding to the `Ticket` object we now have. If none exists, we create one; otherwise we update its counter value.
4. In the fourth and final section we do as the `Value` class: we create the context object, update the `atime` properties of any `Counter` objects we've used, and we render and return the page.

Page templates and CSS

As with exercise #4, we use an HTML page template to render a page for the client. This example is a little more complex and, in particular uses a *style sheet* (`style.css`) to contain all of the effects we apply to the HTML. You do not need to worry about the details here!

`templates/page.html`

```
<!DOCTYPE html>
<html>
  <head>
```

```
  <title>Counting With Tickets Is Even More Fun!</title>
  <meta charset="utf-8" />
  <link href="/static/style.css" rel="stylesheet" type="text/css" />
</head>

<body>
  <p>Time now is {{ now }}.</p>
  {% if ticket %}
  <p>Ticket is {{ ticket.value }}, created at {{ ticket.ctime }}.</p>
  {% endif %}

  {% if counter %}
  <div id="mycounter" class="counter">
    <p>Your latest counter value is {{ counter.value }}.</p>
    <p>Previous access was at {{ counter.atime }}.</p>
    <p>It was created at {{ counter.ctime }}.</p>
  </div>
  {% endif %}

  {% for c in counters %}
  <div class="counter">
    <p>
      Counter:<b>{{ forloop.counter0 }}</b> (<i>{{ c.ticket.value }}</i>)
    </p>
    <p>
      Value:{{ c.value }} Ctime:{{ c.ctime }} Atime:{{ c.atime }}
    </p>
  </div>
  {% endfor %}

  <div class="button">
    <form action="/incr/{% if ticket %}{{ ticket.value }}{% endif %}"
          method="post" accept-charset="utf-8">
      <input id="incr_button" type="submit" value="Increment"/>
    </form>
  </div>

  {% if ticket %}
  <div class="button">
    <form action="/value/{{ ticket.value }}" method="get" accept-charset="utf-8">
      <input id="read_button" type="submit" value="My Values"/>
    </form>
  </div>
  {% endif %}

  <div class="button">
```

```
      <form action="/value" method="get" accept-charset="utf-8">
        <input id="read_button" type="submit" value="All Values"/>
      </form>
    </div>

  </body>
</html>
```

static/style.css

This file is called `style.css` and lives inside a subdirectory of your application called `static`.

```
.button {
    float: left;
}

#mycounter {
    border: 1px solid red;
}

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