

Understanding the Dumper Program

Google Application Engine

University of Michigan – Informatics

This handout describes a very simple and low-level Google Application engine application called “Dumper” that just dumps out the data from a HTTP Request. This application does not demonstrate the pattern we will follow for the real applications – but it does serve to show how things actually work at a low level.

Request / Response Cycle

The basic interaction between a web browser and the web server is that the user clicks on a link (GET request) or submits a form with data (POST request). The browser opens a TCP/IP connection to the server and sends the request (GET or POST) to the server. On the server, the URL is routed to the application and all of the input data (path, parameters, etc) is passed to the application.



The application then runs, possibly accessing a database or some other source of data and then returns the HTTP response to the browser for display.

Reference: <http://en.wikipedia.org/wiki/HTTP>

In this application, we simply dump out the input data from the GET or POST request and return a HTML response that includes a form and the dumped data from the previous request.

The Dumper Program

Note: The sample code for this application can be downloaded from www.appenginelearn.com

The dumper program consists of a very simple **app.yaml** file and a single **index.py** Python file which contains the logic of our App Engine program.

app.yaml:

```
application: ae-02-dumper
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: index.py
```

This simply names out application (ae-02-dumper) and routes all incoming requests to the **index.py** script.

index.py:

```
import os
import sys

print 'Content-Type: text/html'
print ''
print '<form method="post" action="/" >'
print 'Zap Data: <input type="text" name="zap"><br>'
print 'Zot Data: <input type="text" name="zot"><br>'
print '<input type="submit">'
print '</form>'

print '<pre>'
print 'Environment keys:'
print ''
for param in os.environ.keys():
    print param, ': ', os.environ[param]
print ''

print 'Data'
count = 0
for line in sys.stdin:
    count = count + 1
    print line
    if count > 100:
        break

print '</pre>'
```

Note: You will likely end up with problems cutting and pasting source code from PDF handouts into text files. It is better to either download the source code or type it in.

The **index.py** program is broken into three parts:

The first set of print statements simply produce the HTML for a form which can be used to do a test POST of some data to our program.

```
print 'Content-Type: text/html'
print ''
print '<form method="post" action="/" >'
print 'Zap Data: <input type="text" name="zap"><br>'
print 'Zot Data: <input type="text" name="zot"><br>'
print '<input type="submit">'
print '</form>'
```

The first line is a header line that is not part of the HTML data response. Since we are not using a framework, which would send headers for us, we manually produce the header line and then a blank line to indicate the end of the headers and start of the actual HTML data.

The form is quite basic with two text fields and a submit button which will render like this:



The image shows a simple web form. It has two text input fields. The first is labeled 'Zap Data:' and the second is labeled 'Zot Data:'. Below these two fields is a single button labeled 'Submit'.

The next lines of the program read in a set of variables passed in as a Python dictionary. These are the “environment” variables. They are a combination of the server configuration as well as information about the particular request itself.

We simply iterate through the dictionary and then print the items out:

```
print '<pre>'
print 'Environment keys:'
print ''
for param in os.environ.keys():
    print param, ': ', os.environ[param]
print ''
```

The output from this section is as follows:

Environment keys:

```
SERVER_SOFTWARE : Development/1.0
SCRIPT_NAME :
REQUEST_METHOD : GET
PATH_INFO : /
SERVER_PROTOCOL : HTTP/1.0
QUERY_STRING :
CONTENT_LENGTH :
HTTP_USER_AGENT : Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_!
HTTP_CONNECTION : keep-alive
SERVER_NAME : localhost
REMOTE_ADDR : 127.0.0.1
PATH_TRANSLATED : /Users/csev/Desktop/apps/ae-02-dumper/index.py
SERVER_PORT : 8080
AUTH_DOMAIN : gmail.com
CURRENT_VERSION_ID : 1.1
HTTP_HOST : localhost:8080
TZ : UTC
HTTP_CACHE_CONTROL : max-age=0
USER_EMAIL :
HTTP_ACCEPT : text/xml,application/xml,application/xhtml+xml,text
APPLICATION_ID : ae-02-dumper
GATEWAY_INTERFACE : CGI/1.1
HTTP_ACCEPT_LANGUAGE : en-us
CONTENT_TYPE : application/x-www-form-urlencoded
HTTP_ACCEPT_ENCODING : gzip, deflate
```

The environment variables fall into three categories:

- Variables describing the server environment (SERVER_SOFTWARE, SERVER_NAME)
- Variables describing the Request data (REQUEST_METHOD, HTTP_USER_AGENT, or CONTENT_TYPE)
- Variables describing the browser environment variables (HTTP_USER_AGENT, HTTP_ACCEPT, etc.)

Interestingly the documentation about these parameters is described here:

<http://hoohoo.ncsa.uiuc.edu/cgi/in.html>

This is a very “old” web site that describes the Common Gateway Interface (CGI) – which was the way that the very first web servers passed input data from an HTTP request into application code running on the server.

When we are programming at this level (which we will not do for long) – we are using the old mystical ways of the early world-wide-web. We won’t use this

program pattern for much longer – but it is good to start by understanding the low-level details and then delegate the handling of those details to a web framework.

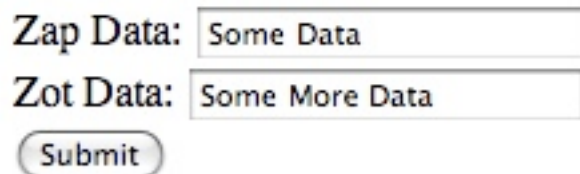
The last part of the **index.py** program dumps out up to the first 100 lines of POST data if the data exists:

```
print 'Data'
count = 0
for line in sys.stdin:
    count = count + 1
    print line
    if count > 100:
        break
```

According to the rules of Common Gateway Interface – the POST data is simply presented to the application on its “Standard Input”. In Python we can read through the predefined file handle **sys.stdin** to access our POST data.

If you look at the bottom of the initial output of the program – you will see that there is no POST data- because when you navigate to **http://localhost:8080** the browser issues a GET request for the initial page.

To test POST data dumping code, we must enter some data into the Zap and Zot data fields and press Submit:



When we press “Submit”, our browser sends a POST request – which you can immediately see in the REQUEST_METHOD variable change from GET to POST:

Environment keys:

```
HTTP_REFERER : http://localhost:8080/
SERVER_SOFTWARE : Development/1.0
SCRIPT_NAME :
REQUEST_METHOD : POST
PATH_INFO : /
```

And if we scroll down to the bottom of the output, you can see the actual POST data:

```
GATEWAY_INTERFACE : CGI/1.1
HTTP_ACCEPT_LANGUAGE : en-us
CONTENT_TYPE : application/x-www-form-urlencoded
HTTP_ACCEPT_ENCODING : gzip, deflate
```

```
Data
zap=Some+Data&zot=Some+More+Data
```

The POST data is encoded by escaping spaces and special characters. We would have to parse the input data using string parsing and then un-escape the data to get back to the actual data that was typed into the form.

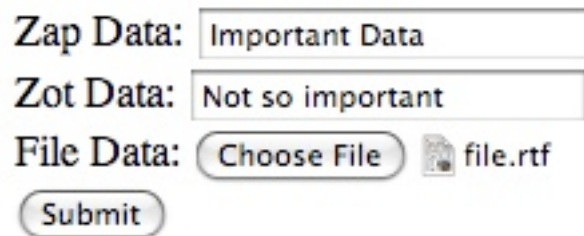
Thankfully – in our next application, a web framework will soon do all that parsing and escaping for us.

Advanced: Uploading Files

If you want to play a bit – you can experiment with how files are uploaded in the HTTP request/response cycle. Make the following changes to the form code:

```
print '<form method="post" action="/" enctype="multipart/form-data">'
print 'Zap Data: <input type="text" name="zap"><br>'
print 'Zot Data: <input type="text" name="zot"><br>'
print 'File Data: <input type="file" name="filedat"><br>'
print '<input type="submit">'
```

Then run the program again, selecting a file, typing some data and then pressing submit:



The screenshot shows a web form with three rows of input fields. The first row is labeled 'Zap Data:' and contains a text input field with the value 'Important Data'. The second row is labeled 'Zot Data:' and contains a text input field with the value 'Not so important'. The third row is labeled 'File Data:' and contains a file input field with a 'Choose File' button and a file icon next to the text 'file.rtf'. Below these fields is a 'Submit' button.

If you look at the output from the program it looks as follows:

```
CONTENT_TYPE : multipart/form-data; boundary=----WebKitFormBoundaryJ6xgTm1AiTZSKBYD
HTTP_ACCEPT_ENCODING : gzip, deflate
```

Data

```
-----WebKitFormBoundaryJ6xgTm1AiTZSKBYD
```

```
Content-Disposition: form-data; name="zap"
```

Important Data

```
-----WebKitFormBoundaryJ6xgTm1AiTZSKBYD
```

```
Content-Disposition: form-data; name="zot"
```

Not so important

```
-----WebKitFormBoundaryJ6xgTm1AiTZSKBYD
```

```
Content-Disposition: form-data; name="filedat"; filename="file.rtf"
```

```
Content-Type: text/rtf
```

```
{\rtf1\ansi\ansicpg1252\cocoartf949\cocoasubrtf350
```

```
{\fonttbl\f0\fswiss\fcharset0 Helvetica;}
```

```
{\colortbl;\red255\green255\blue255;}
```

The content type change changed to “**multipart/form-data**” and now the form values and the file data are spread out in the input stream with this complex looking separator that divides all the parts.

This is necessary because the file will be a lot of data – and it will come in as many lines. Also there needs to be a lot of description about the nature of the data.

Reference: <http://tools.ietf.org/html/rfc2046#section-5.1.3>

Also if you are looking at the log from your application you will see entries as follows:

```
INFO      2008-10-20 13:09:21,151 dev_appserver.py] "GET / HTTP/1.1" 200
INFO      2008-10-20 13:09:23,374 dev_appserver.py] "GET / HTTP/1.1" 200
INFO      2008-10-20 13:09:30,463 dev_appserver.py] "POST / HTTP/1.1"
200 -
```

You can see the different GET and POST requests begin processed in the App Engine log.

Summary

This simple program allows us to look at how the HTTP Request/Response style is supported in the Google Application Engine. The Application Engine framework gives us a very primitive Common Gateway (GCI) compliant interface with environment variables, standard input, and standard output. We can examine all the data that the App Engine makes available to our scripts.

Ultimately this is only of passing interest because we will delegate much of the detail of handling the request and response to the built in Web Application framework in the Google Application Engine.

When we use the framework, our code may seem a little more complex but the framework takes care of a myriad of small details of parameter passing, parsing, headers, and conversion.

Reference:

<http://code.google.com/appengine/docs/gettingstarted/usingwebapp.html>

This materials is Copyright Creative Commons Attribution 2.5 – Charles Severance

Comments and questions to csev@umich.edu www.dr-chuck.com