

Using the webapp Framework

Google Application Engine

University of Michigan - Informatics

This handout describes the basics of the Google Application Engine web application framework – webapp. The webapp library takes care of many of the mundane details of the Hypertext Transport Protocol (HTTP) interactions. The webapp framework handles all details like parameter parsing, multiple parameter formats, etc.

While code using the webapp framework may look a little more complex than the lower-level code looking at the variables and input directly, in the long run your web applications will be far smaller and fully compliant with the subtle details HTTP protocol – they won't break because you missed some small detail in the protocol that you did not notice until some user started using a different browser than the ones that you used for testing.

A Basic WebApp Application

The following is a basic web application (ae-03-webapp) :

```
import logging
import wsgiref.handlers
from google.appengine.ext import webapp

class MainHandler(webapp.RequestHandler):

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

    def get(self):
        logging.info("Hello GET")
        self.dumper()

    def post(self):
        logging.info("Hello POST")
        self.dumper()

    def dumper(self):
        self.response.out.write(self.formstring)
        self.response.out.write("<pre>\n")
        self.response.out.write('Request parameters:\n')
        for key in self.request.params.keys():
            value = self.request.get(key)
            if len(value) < 100:
                self.response.out.write(key+' : '+value+'\n')
```

```

        else:
            self.response.out.write(key+':'+str(len(value))+
                ' (bytes long)\n')
        self.response.out.write('\n')

def main():
    application = webapp.WSGIApplication([
        ('/*.*', MainHandler)],
                                         debug=True)
    wsgiref.handlers.CGIHandler().run(application)

if __name__ == '__main__':
    main()

```

The first thing that you notice is the use of the main program pattern. There is a **main()** function that is defined which is called when the code is running as a main program (i.e. running as a web server). This pattern is primarily used to keep the **main()** code from running if this file were imported into some other bit of Python – perhaps a testing framework.

When the request does come in the code is running as a main program so Python parses the entire file and then runs the code in **main()**.

The code in main:

```

def main():
    application = webapp.WSGIApplication([
        ('/*.*', MainHandler)],
                                         debug=True)
    wsgiref.handlers.CGIHandler().run(application)

```

Is setting up several things – it is creating WSGIApplication object in the variable application and then “starting” the web application up. This allows the webapp framework to handle the incoming request.

When the run(application) is called, the **wsgi** system starts up, takes a look at the incoming request and decides what to do. We give the framework a routing table in the form of:

```
[ ('/*.*', MainHandler) ]
```

This is a list indicated by square brackets [] of tuples indicated by parenthesis () where each tuple consists of a pattern to match in the URL and a bit of code to call (**MainHandler**) when the pattern matches. Translated to English, that this means is “send all urls to the MainHandler”. We do this because the program is simpler.

It might be easier to look at a later program with two URL handlers its call to WSGIApplication looks as follows:

```
application = webapp.WSGIApplication([
```

```
('/grades', GradeHandler),
('/.*', MainHandler)],
                                debug=True)
```

Translating this routing list to English, it is saying, “route urls of the form /grades to the GradeHandler and all the rest of the urls to the MainHandler”.

So this list of tuples gives the webapp framework an indication of where to call us back depending on the pattern of the URL – this is a URL routing table.

Back to our example, we are telling the framework to call us back once it has looked at the HTTP request using the MainHandler code that we provide:

```
def main():
    application = webapp.WSGIApplication([
        ('/.*', MainHandler)],
                                        debug=True)
    wsgiref.handlers.CGIHandler().run(application)
```

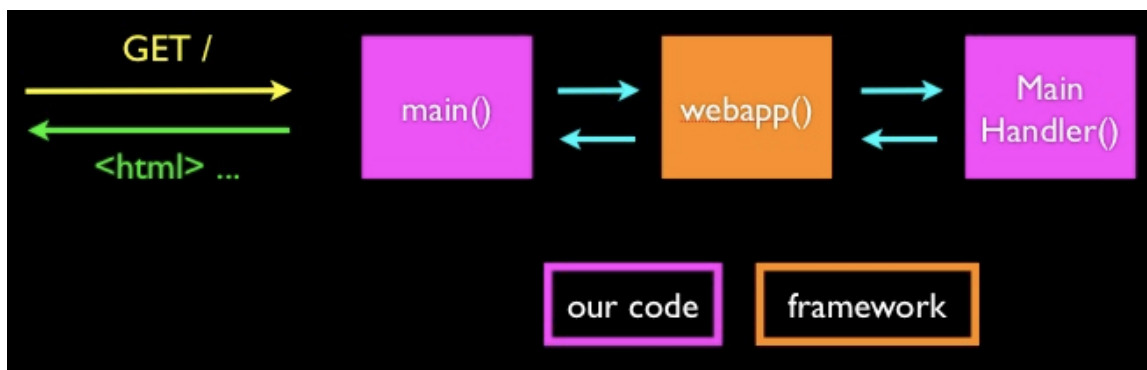
The Call Back Pattern

The call-back pattern is very common in Object Oriented programming. The basic idea is that we hand over the main responsibility for handling something to a framework (i.e. a large amount of code we did not write) – and then let the framework call one of our objects back – at some important moment when it wants us to participate in the handling of the request.

This pattern is used in many situations ranging from graphical user interfaces to message/event handling. We initially communicate to the framework those “happenings” or event that we are interested in and give the framework a bit of our code to call to “handle” those events.

That is why we use the convention of naming these bits of code with “Handler” in their names – they are designed to “Handle” something.

The pattern is as follows:



The incoming HTTP request arrives to our main program. Instead of handling the request directly, we simply set up the framework and tell it under what conditions (urls that match `/*`) and where (MainHandler) to call us back when it needs some “assistance” from us.

Then the framework starts up and looks at the HTTP request, figuring out which kind of request it is – parsing all of the data, converting file input if necessary – and then calls out **MainHandler** – using either the `get()` or `post()` method as appropriate.

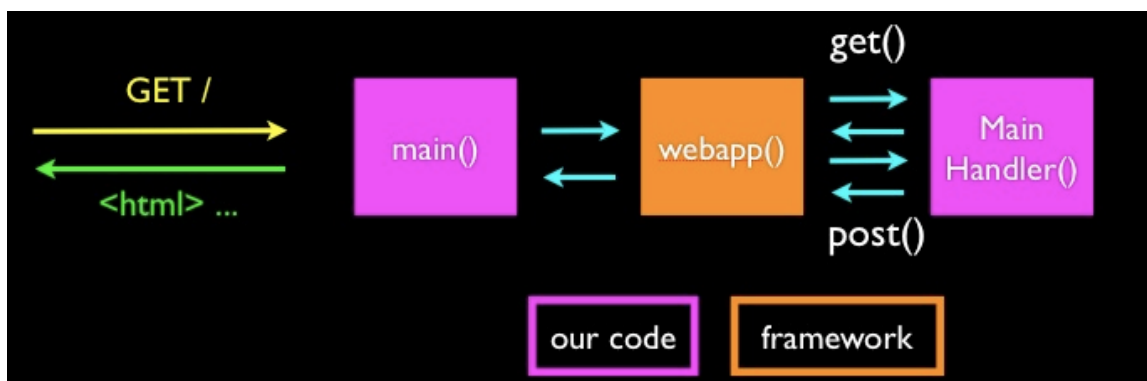
Looking at the Handler Code

First, we will look at a trivial variant of the MainHandler to get a sense of the basic structure of handlers:

```
class MainHandler(webapp.RequestHandler):  
  
    def get(self):  
        logging.info("Hello GET")  
  
    def post(self):  
        logging.info("Hello POST")
```

The first thing to notice is that the MainHandler extends the **webapp.RequestHandler** class – this means that our handler inherits a lot of functionality from this class – and whatever we do in our class is in addition to the functionality in the **webapp.RequestHandler** class.

There are two methods which we provide to the framework in our Handler class – a `get()` method and a `post()` method. The framework will look at the incoming request and call the proper method for us. If the request is a GET request the framework will call the `get()` method and if the request is a POST request, the framework will call our `post()` method. The framework saves us from figuring out which kind of request we have.



We should be making up a response to the request in our methods – but since this is a trivial handler, all we do is issue a log message.

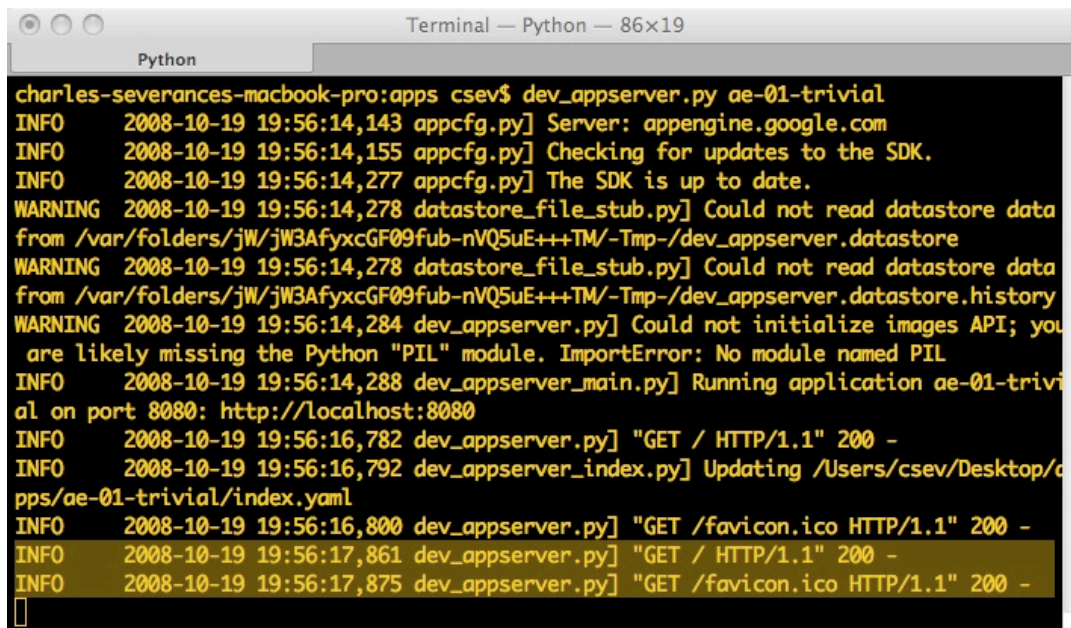
Web Server Logs

Since our software takes incoming HTTP Requests and Produces an HTTP response, often in some end-user's browser half-way around the world, it is a little hard to figure out what happened when something goes wrong. Usually if your program fails, the user will shake their head and switch to another web site in disgust.

They will never call you and talk for a while about what they did that went wrong and what strange messages they saw – frankly – you do not want them calling you at all hours of the night when they encounter an error – you would be far happier if in the morning – you could see what went wrong over night and get some detail as to what went wrong overnight.

This is the purpose of a “log”. A log is generally a file or window that contains messages from your program.

You have been using logs all along – this is an example log:



```
Terminal — Python — 86x19
Python
charles-severances-macbook-pro:apps csev$ dev_appserver.py ae-01-trivial
INFO      2008-10-19 19:56:14,143 appcfg.py] Server: appengine.google.com
INFO      2008-10-19 19:56:14,155 appcfg.py] Checking for updates to the SDK.
INFO      2008-10-19 19:56:14,277 appcfg.py] The SDK is up to date.
WARNING   2008-10-19 19:56:14,278 datastore_file_stub.py] Could not read datastore data
from /var/folders/jw/jW3AfyxcGF09fub-nVQ5uE+++TM/-Tmp-/dev_appserver.datastore
WARNING   2008-10-19 19:56:14,278 datastore_file_stub.py] Could not read datastore data
from /var/folders/jw/jW3AfyxcGF09fub-nVQ5uE+++TM/-Tmp-/dev_appserver.datastore.history
WARNING   2008-10-19 19:56:14,284 dev_appserver.py] Could not initialize images API; you
are likely missing the Python "PIL" module. ImportError: No module named PIL
INFO      2008-10-19 19:56:14,288 dev_appserver_main.py] Running application ae-01-triv
al on port 8080: http://localhost:8080
INFO      2008-10-19 19:56:16,782 dev_appserver.py] "GET / HTTP/1.1" 200 -
INFO      2008-10-19 19:56:16,792 dev_appserver_index.py] Updating /Users/csev/Desktop/a
pps/ae-01-trivial/index.yaml
INFO      2008-10-19 19:56:16,800 dev_appserver.py] "GET /favicon.ico HTTP/1.1" 200 -
INFO      2008-10-19 19:56:17,861 dev_appserver.py] "GET / HTTP/1.1" 200 -
INFO      2008-10-19 19:56:17,875 dev_appserver.py] "GET /favicon.ico HTTP/1.1" 200 -
```

When the AppEngine server is running the log streams out to the window in which you started the AppServer. When you upload your application to the Google Infrastructure – it still maintains a log that you can check in a browser.

simplelti ▾ Version: 1 [« Show All Applications](#)

[Dashboard](#)
[Logs](#)
 Datastore
[Indexes](#)
[Data Viewer](#)
 Administration
[Application Settings](#)
[Developers](#)
[Versions](#)

Filter Logs [?](#)
 Minimum Severity: **Error** ▾ [+ Options](#)

Tip: Click a log line to show or hide its details. [+ Expand logs](#)

+	10-17 10:07AM 55.555	/upload	500	85ms	276mcycles	0kb
E	10-17 10:07AM 55.636	Traceback (most recent call last): File "/base/python_lib/versions/1/google/				
+	10-17 08:40AM 26.858	/upload	500	22ms	59mcycles	0kb
E	10-17 08:40AM 26.877	Traceback (most recent call last): File "/base/python_lib/versions/1/google/				
+	10-14 10:08PM 17.189	/upload	500	308ms	995mcycles	0kb
E	10-14 10:08PM 17.489	Traceback (most recent call last): File "/base/python_lib/versions/1/google/				

So you can look at the log of a running web application any time and see what is going wrong. You can see both successful activities in the log and get a sense of patterns of interaction as well as seeing errors in the log such as the following:

```
Terminal — bash — 81x19
bash
charles-severances-macbook-pro:apps csev$ dev_appserver.py ae-01-trivial
ERROR 2008-10-19 19:33:37,013 dev_appserver_main.py] Fatal error when loading
application configuration:
Invalid object:
Unknown url handler type.
<URLMap
  static_dir=None
  secure=never
  script=None
  url=/. *
  static_files=None
  upload=None
  expiration=None
  login=optional
  mime_type=None
>
in "ae-01-trivial/app.yaml", line 8, column 1
charles-severances-macbook-pro:apps csev$
```

After a while – you will get used to the logs and their patterns and rhythms – once you become familiar with your application – it is almost like watching the screens in the Matrix – after a while – it just starts to make sense to you.

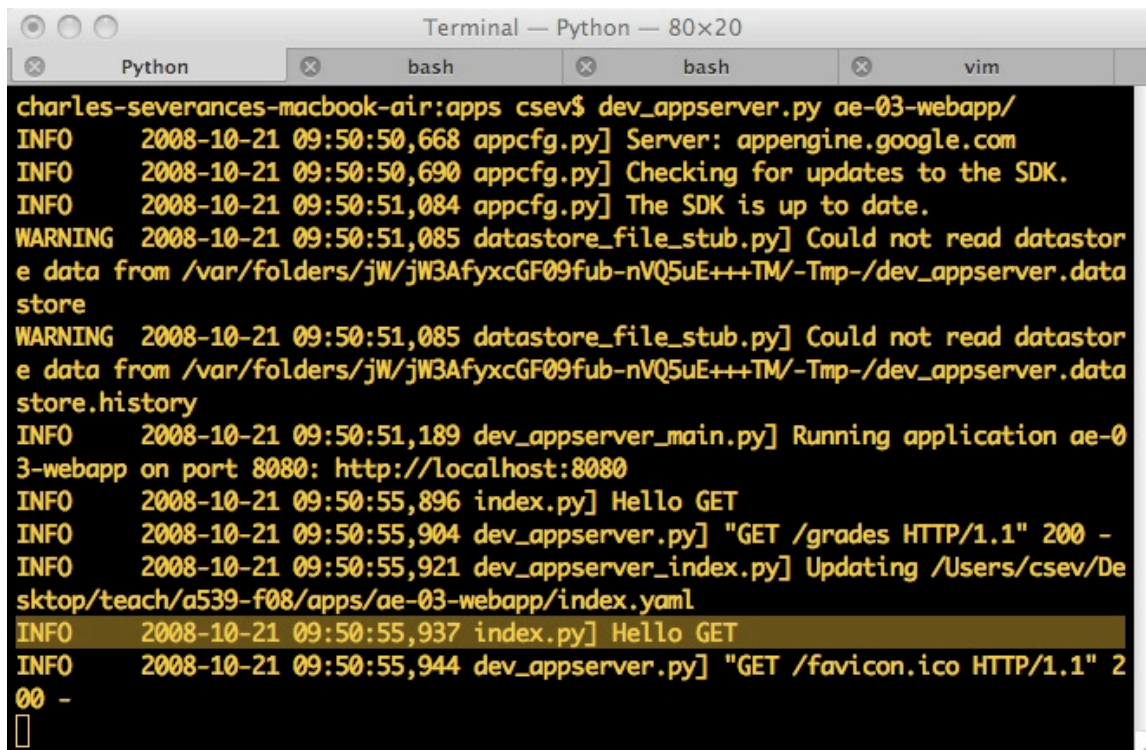
Back here in the real world, we have our trivial Handler code:

```
class MainHandler(webapp.RequestHandler):
    def get(self):
        logging.info("Hello GET")
```

```
def post(self):
    logging.info("Hello POST")
```

All this code does is log a message (at the info logging level) that indicates whether this is a POST or a GET – this is a good way to make sure things are routed to the right place.

When the program is executing you will see the log entries popping up as shown below:

A terminal window titled "Terminal — Python — 80x20" with tabs for "Python", "bash", "bash", and "vim". The terminal output shows the following log entries:

```
charles-severances-macbook-air:apps csev$ dev_appserver.py ae-03-webapp/
INFO    2008-10-21 09:50:50,668 appcfg.py] Server: appengine.google.com
INFO    2008-10-21 09:50:50,690 appcfg.py] Checking for updates to the SDK.
INFO    2008-10-21 09:50:51,084 appcfg.py] The SDK is up to date.
WARNING 2008-10-21 09:50:51,085 datastore_file_stub.py] Could not read datastor
e data from /var/folders/jW/jW3AfyxcGF09fub-nVQ5uE+++TM/-Tmp-/dev_appserver.data
store
WARNING 2008-10-21 09:50:51,085 datastore_file_stub.py] Could not read datastor
e data from /var/folders/jW/jW3AfyxcGF09fub-nVQ5uE+++TM/-Tmp-/dev_appserver.data
store.history
INFO    2008-10-21 09:50:51,189 dev_appserver_main.py] Running application ae-0
3-webapp on port 8080: http://localhost:8080
INFO    2008-10-21 09:50:55,896 index.py] Hello GET
INFO    2008-10-21 09:50:55,904 dev_appserver.py] "GET /grades HTTP/1.1" 200 -
INFO    2008-10-21 09:50:55,921 dev_appserver_index.py] Updating /Users/csev/De
sktop/teach/a539-f08/apps/ae-03-webapp/index.yaml
INFO    2008-10-21 09:50:55,937 index.py] Hello GET
INFO    2008-10-21 09:50:55,944 dev_appserver.py] "GET /favicon.ico HTTP/1.1" 2
00 -
█
```

This allows you to track your program internally and is particularly helpful when things are not going well – you can put a message in the log when you encounter strange or error conditions.

Since the user does not see the log – you can add plenty of detail to help you figure out the source of the problem – particularly given that you will be looking at the log hours after the actual error occurred in your application.

You can see why some of the error messages err on the side of verbosity and provide far more detail than you might need – because you often need to reconstruct what happened – only by looking at the log.

Reference: <http://code.google.com/appengine/articles/logging.html>

Looking at the Real Handler Code

The actual handler code does more than just placing a message in the log. The `get()` and `post()` methods actually look as follows:

```
class MainHandler(webapp.RequestHandler):

    def get(self):
        logging.info("Hello GET")
        self.dumper()

    def post(self):
        logging.info("Hello POST")
        self.dumper()
```

After they put a friendly message in the log, they call another method in the same class called **dumper()**. We call `dumper` regardless of whether or not the request is a GET or POST – because we just want to dump everything.

The code for `dumper` is as follows:

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

    def dumper(self):
        self.response.out.write(self.formstring)
        self.response.out.write("<pre>\n")
        self.response.out.write('Request parameters:\n')
        for key in self.request.params.keys():
            value = self.request.get(key)
            if len(value) < 100:
                self.response.out.write(key+' :'+value+'\n')
            else:
                self.response.out.write(key+' :'+str(len(value))+
                    ' (bytes long)\n')
        self.response.out.write('\n')
```

Don't be alarmed by the syntax of the **formstring** assignment statement. Python uses triple quotes (""") to indicate a string that can cross line boundaries – the string continues until a closing triple quote. This allows us several lines of text into a single string.

In the **dumper()** code – we don't use the `print` statement – instead we call **self.response.out.write()** and pass it a string for each bit of output we are producing – this is complex syntax for a simple concept.

Instead of writing directly to the output using print, we are to hand our response text back to the webapp framework in the **self.response** object. By doing this we allow the Google Application Engine some flexibility in how it actually handles the request and its output.

Looking through the code – the first thing we do is add a form to the response from the **formstring**. Then we loop through the request parameters using a dictionary of the input data in **self.request.parms**.

We check to see how long the parameter value is and only print out the length of the data for longer parameter values. This is because the webapp framework has completely handled the parsing of normal or multi-part form encoded data including automatically converting any uploaded files that are part of the form.

Here is what the screen looks like after a GET request to `http://localhost:8080/`

Zap Data:
Zot Data:
File Data: no file selected

Request parameters:

Then we will in the form with some data:

Zap Data:
Zot Data:
File Data: no file selected

Request parameters:

And press Submit:

Zap Data:
Zot Data:
File Data: no file selected

Request parameters:
zap:Some Data
zot:Some More Data
filedat:

To see the data printed out.

If we select a file and upload it, the output looks as follows:

```
Zap Data:   
Zot Data:   
File Data:  no file selected  
  
  
Request parameters:  
zap:This Time  
zot:We put in a File  
filedat:388 (bytes long)
```

You can see that the webapp framework converted the multipart form data and has just handed us the contents of the file in the **filedat** parameter.

So while the code seems initially more complex, if you recall the complexity of multipart form data from our raw dumper program, you can appreciate the value of using the webapp framework to handle the details of parsing and converting the incoming requests for us.

```
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;}
```

Summary

The Google Application Engine webapp framework moves us towards an object-oriented approach to handling out HTTP requests and responses. We initially set things up by creating a web application and giving it a routing table to call us back to handle the incoming requests.. Then the framework parses and interprets the incoming request and calls the correct methods in our handler code to process the input data and prepare the HTTP response.

We also took a look at how application logs are used in a web application to help you monitor what is happening when your application is running and potentially experiencing errors – and you are not in contact with the ultimate end-users of your application.

This materials is Copyright Creative Commons Attribution 2.5 – Charles Severance

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