Google Application Engine
Templates for HTML

University of Michigan – Informatics
Charles Severance

While it is possible to generate all of the HMTL of your application from within strings in the Python, this is generally a poor way to author HTML. In particular, it means that every time you want to change a bit of the generated HTML you need to dig through the program, find the strings and then change the HTML code:

```
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>"""
```

At the same time, our web application needs to have some parts of the web pages be generated dynamically as part of the code of the web application – either based on the user's input or based on some information retrieved from the database.

The compromise for this is to introduce the notion of a "template". A template is a file that contains mostly HTML – but there are specially marked areas of the template that is replaced by data handed to the template from the Python code when the template is to be rendered.

There are many different templating languages and syntaxes. The default template syntax used by the Google Application Engine is from the Django project.

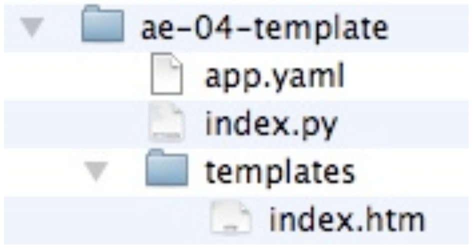
Template Syntax

The template syntax in Google Application augments the HTML using curly braces to identify template directives. The template for our dumper program:

```
<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>
<pre>
Request Data:
{{ dat }}
</pre>
```

The area in the template that will be replaced with data from Python is the area in double curly braces. In between the double curly braces "dat" is a key that is used to determine which piece of data from the Python code to put into the template to replace the **{{ dat }}** in the template.

By convention, we put the templates into a directory named "templates". This way we can easily keep HTML templates separate from the Python code.



Naming the folder "templates" is not a rule – it is a convention. It is a good convention because it means that other developers will immediately know where to find the templates.

Using the Templates from Python

To display the template in Python we add code to "render" the template and then print the output of the render process to the HTTP response. Here is some simple code which accomplishes renders the index.htm file:

```
import os
from google.appengine.ext.webapp import template

temp = os.path.join(os.path.dirname(__file__), 'templates/index.htm')
outstr = template.render(temp, {'dat': ‘Hello There’})
self.response.out.write(outstr)
```

The first line is a bit complex looking – it calls the Python operating system library (**os**) to look up the location of the **index.htm** file in the folder named **templates**.

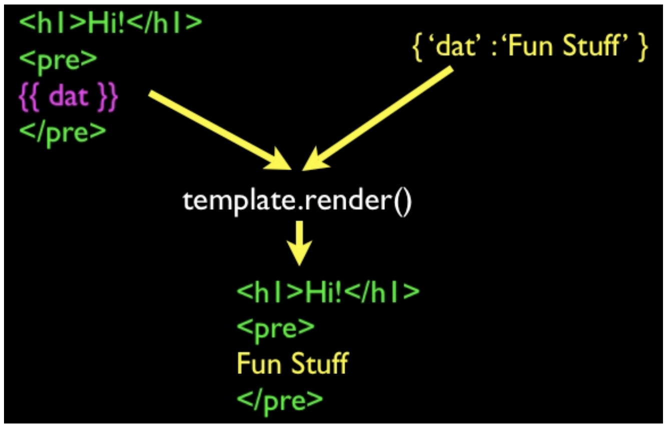
The real work is done in the **template.render()** line. This takes two parameters – the first is the location of the template file and the second is a Python dictionary object which contains the strings to be placed in the template where the **{{ dat }}** entries are found. The results of the substitution of the variables into the template are returned as a string in the variable **outstr**.

Then the string that (**outstr**) is returned from the render operation is then written out as the HTTP response. The text returned from the render process will look as follows:

```
<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>
<pre>
Request Data:
Hello There
</pre>
```

The data from the Python dictionary is now substituted into the template as it is rendered. The following is a diagram of the process.



The process is simple – render engine looks for the "hot spots" in the template and when it finds a spot where a substitution is required, the renderer looks in the Python dictionary to find the replacement text.

The template language is actually quite sophisticated – we will look at the capabilities of the template language in more detail later.

Overall Flow

To go back to our example and put it all in context, we can see how our dumper program has evolved:

```
def dumper(self):
  prestr = ' '
  for key in self.request.params.keys():
    value = self.request.get(key)
    if len(value) < 100:
       prestr = prestr + key+':'+value+'\n'
    else:
       prestr = prestr + key+':'+str(len(value))+' (bytes long)\n'
  temp = os.path.join(os.path.dirname(__file__), 'templates/index.htm')
  outstr = template.render(temp, {'dat': prestr})
  self.response.out.write(outstr)
```

The first half of the **dumper()** method loops through the request parameters and instead of writing the parameter information to the response, the information is appended into the **predat** string including labeling information, the parameters themselves and newlines ("\n").

Then the **predat** string is then passed into the **template.render()** as in a dictionary under the key "**dat**".

The render engine then merges the data from the Python code with the HTML template and returns the resulting HTML that is written out as the response.

Abstraction and Separation of Concerns – "Model View Controller"

For complex web programs it is very important to be organized – and for each area of functionality to have its place. Following commonly understood patterns helps us keep track of the bits of the program as it becomes increasingly complex.

One of the most common programming patterns in web-based applications is called "Model-View-Controller" or **MVC** for short. Many web frameworks such as Ruby on Rails and Spring MVC follow the Model-View-Controller pattern.

The MVC pattern breaks the code in a web application into three basic areas:

- **Controller** - The code that does the thinking and decision making

- **View** - The HTML, CSS, etc. which makes up the look and feel of the application

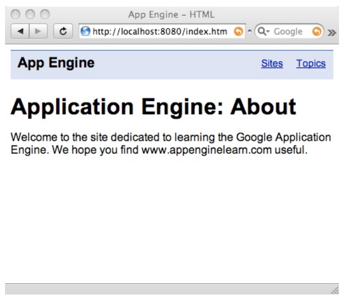
- **Model** - The persistent data that we keep in the data store

In a Google Application Engine program, the **index.py** is an example of **Controller** code and the HTML in the **templates** is an example of a **View**.

We will encounter the **Model** in a later chapter and then we will revisit the MVC pattern and explore it in more detail.

Multiple Templates

Real applications have many different views (screens) so now we will explore how to support multiple templates in our application. We will build a simple application which has three screens and navigation between the screens.

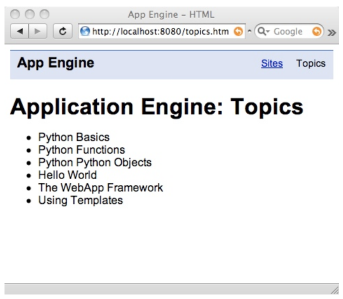

App Engine – HTML

http://localhost:8080/index.htm

App Engine Sites Topics

Application Engine: About

Welcome to the site dedicated to learning the Google Application Engine. We hope you find www.appenginelearn.com useful.

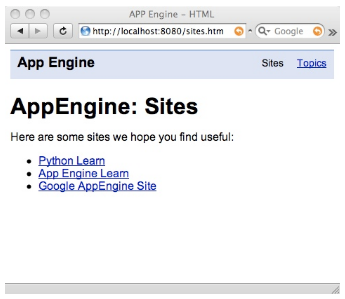

App Engine – HTML

http://localhost:8080/topics.htm

App Engine Sites Topics

Application Engine: Topics

- Python Basics
- Python Functions
- Python Python Objects
- Hello World
- The WebApp Framework
- Using Templates


APP Engine – HTML

http://localhost:8080/sites.htm

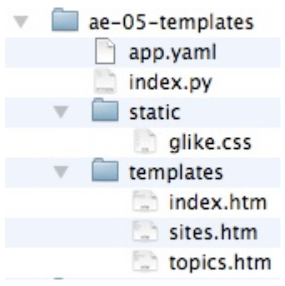
App Engine Sites Topics

AppEngine: Sites

Here are some sites we hope you find useful:

- Python Learn
- App Engine Learn
- Google AppEngine Site

This application uses CSS to make the navigation appear at the top of the screen. Looking at the application layout, we see one template for each HTML page.


```
ae-05-templates
    app.yaml
    index.py
    static
        glike.css
    templates
        index.htm
        sites.htm
        topics.htm
```

The **templates** folder contains the **templates** that are used to create output dynamically (rendered) in the Python code (**index.py**) – the **static** folder contains files that do not change – they may be CSS, Javascript, or image files.

Static Files

We indicate that the **static** folder holds these “un-changing” files by adding an entry to the **app.yaml** file:

```
application: ae-05-templates
version: 1
runtime: python
api_version: 1

handlers:
- url: /static
  static_dir: static

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

We add a new handler entry with the special indicator to map URLs which start with **/static** to the special **static** folder. And we indicate that the material in the **static** folder is non-dynamic.

The order of the URL entries in the handler section is important – it first checks to see if an incoming URL starts with “/static” – if there is a match – the content is served from the static folder. If there is no match, the "catch-all" URL (/.*) routes all URLs to the **index.py** script.

The convention is to name the folder "**static**". You technically could name the folder and path (**/static**) anything you like. The **static_dir** directive (like the script directive) is a directive and cannot be changed. But the simple thing is to always follow the pattern and name the folder and path "**static**".

The advantage of placing files in a static folder is that it does not use your program (**index.py**) for serving these files. Since you may be paying for the CPU usage of the Google servers – avoiding CPU usage on serving static content is a good idea. The static content still counts against your data transferred – it just does not incur CPU costs.

Even more importantly, when you indicate that files are static, it allows Google to distribute these files to many different servers geographically and leave the files there. This means that retrieving these files from different continents may be using Google servers closest to the user of your application. This means that your application can scale to far more users efficiently.

Referencing Static Files

Once you put a file in the static folder, you simply reference those files with absolute paths which start with /static as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
   <title>App Engine - HTML</title>
   <link href="/static/glike.css" rel="stylesheet" type="text/css" />
 </head>
 <body>
   <div id="header">
```

When the AppEngine sees the URL, it routes it to one of its many distributed copies of the file and serves up the content.

Generalizing Template Lookup with Multiple Templates

In the above example (ae-04-template) there was only one template so we hard-coded the name of the template when we wanted to do the render operation. Sometimes you are in a handler that knows the exact name of the template so it can follow that pattern. Other times, you want to have a bunch of templates and serve them up with paths such as:

```
      http://localhost:8080/index.htm
      http://localhost:8080/topics.htm
      http://localhost:8080/sites.htm
```

And we do not want to have to write special code to look up each template separately. We can create general purpose code to lookup a template based on the incoming path of the request – and if we find a matching template, we use that template and otherwise we use the index.htm template.

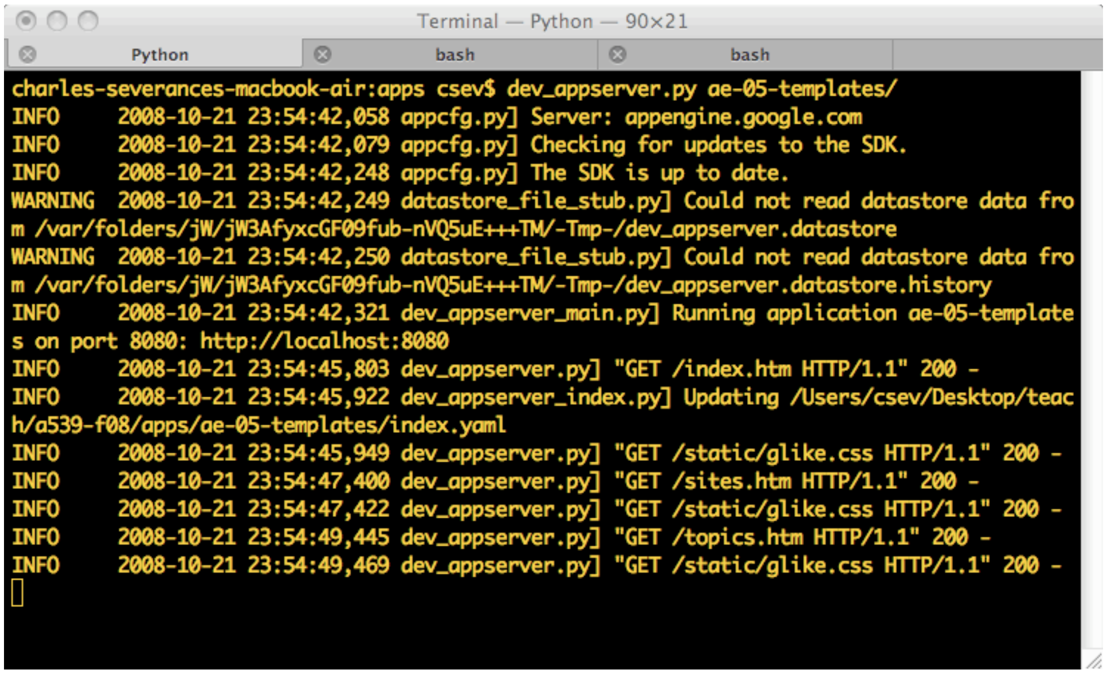
Here is the controller code to accomplish this:

```
class MainHandler(webapp.RequestHandler):

  def get(self):
    path = self.request.path
    try:
        temp = os.path.join(os.path.dirname(__file__), 'templates' + path)
        outstr = template.render(temp, { })
        self.response.out.write(outstr)
    except:
        temp = os.path.join(os.path.dirname(__file__), 'templates/index.htm')
        outstr = template.render(temp, { })
        self.response.out.write(outstr)
```

We first pull in the path from **self.request.path** and append the path to templates – we try to perform a render using this file – and if it fails, we go to the **except:** processing and render using the **index.htm** template.

The **self.request.path** variable shows the path within the application that is being requested. In the log output below, you can see each path being requested using a GET request.



```
charles-severances-macbook-air:apps csev$ dev_appserver.py ae-05-templates/
INFO     2008-10-21 23:54:42,058 appcfg.py] Server: appengine.google.com
INFO     2008-10-21 23:54:42,079 appcfg.py] Checking for updates to the SDK.
INFO     2008-10-21 23:54:42,248 appcfg.py] The SDK is up to date.
WARNING  2008-10-21 23:54:42,249 datastore_file_stub.py] Could not read datastore data fro
m /var/folders/jW/jW3AfyxcGF09fub-nVQ5uE+++TM/-Tmp-/dev_appserver.datastore
WARNING  2008-10-21 23:54:42,250 datastore_file_stub.py] Could not read datastore data fro
m /var/folders/jW/jW3AfyxcGF09fub-nVQ5uE+++TM/-Tmp-/dev_appserver.datastore.history
INFO     2008-10-21 23:54:42,321 dev_appserver_main.py] Running application ae-05-template
s on port 8080: http://localhost:8080
INFO     2008-10-21 23:54:45,803 dev_appserver.py] "GET /index.htm HTTP/1.1" 200 -
INFO     2008-10-21 23:54:45,922 dev_appserver_index.py] Updating /Users/csev/Desktop/teac
h/a539-f08/apps/ae-05-templates/index.yaml
INFO     2008-10-21 23:54:45,949 dev_appserver.py] "GET /static/glike.css HTTP/1.1" 200 -
INFO     2008-10-21 23:54:47,400 dev_appserver.py] "GET /sites.htm HTTP/1.1" 200 -
INFO     2008-10-21 23:54:47,422 dev_appserver.py] "GET /static/glike.css HTTP/1.1" 200 -
INFO     2008-10-21 23:54:49,445 dev_appserver.py] "GET /topics.htm HTTP/1.1" 200 -
INFO     2008-10-21 23:54:49,469 dev_appserver.py] "GET /static/glike.css HTTP/1.1" 200 -
```

You can see the browser requesting a path like "/sites.htm" which renders from the template and then when the browser sees the reference to the "/static/glike.css" the browser then does another GET request to retrieve the CSS which is stored in the static folder.

The **self.request.path** starts with a slash (/) so when it is appended to "templates" the path we hand to the render engine looks like

```
templates/sites.htm
```

Which is exactly where we have stored our template.

Extending Base Templates

We have only begun to scratch the surface of the capabilities of the templating language. Once we have successfully separated our views from the controller, we can start looking at ways to manage our views more effectively.

If you look at the example HTML files used as templates in this application, you will find that the files are nearly identical except for a few small differences between each file. Most of the content of the file is identical and copied between files.

```
<head>
   <title>App Engine - HTML</title>
   <link href="/static/glike.css" rel="stylesheet" type="text/css" />
 </head>
 <body>
   <div id="header">
      <h1><a href="index.htm" class="selected">
              App Engine</a></h1>
      <ul class="toolbar">
        <li><a href="sites.htm">Sites</a></li>
        <li><a href="topics.htm" class="selected">Topics</a></li>
      </ul>
   </div>
   <div id="bodycontent">
      <h1>Application Engine: Topics</h1>
      <ul>
        <li>Python Basics</li>
        <li>Python Functions</li>
        <li>Python Python Objects</li>
        <li>Hello World</li>
        <li>The WebApp Framework</li>
        <li>Using Templates</li>
      </ul>
   </div>
 </body>
</html>
```

The only things that change between the files is which link is selected (i.e. **class="selected"**) and the information in the "**bodycontent**" div. All the material in the <head> area and nearly all material in the **header** div are identical between files.

We cause a significant maintenance problem when we repeat this text in many (perhaps hundreds) files in our application. When we want to make a change to this common content – we have to edit all the files and make the change – this becomes tedious and error prone. It also means that we have to test each screen separately to make sure it is updated and working properly.

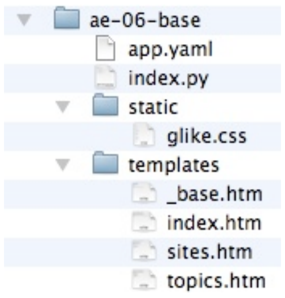
To solve this, we create a special template that contains the common material for each page. Then the page files only include the material that is different. Here is a sample page file that is making use of the base template. Here is the new **index.htm** template file:

```
{% extends "_base.htm" %}
{% block bodycontent %}
      <h1>Application Engine: About</h1>
      <p>
      Welcome to the site dedicated to
      learning the Google Application Engine.
      We hope you find www.appenginelearn.com useful.
      </p>
{% endblock %}
```

The templating language uses curly braces to indicate our commands to the render engine. The first line of says this page starts with the text contained in the file "**_base.htm**". We are starting with **_base.htm** and *extending* it.

The second line says, "when you find an area marked as the "**bodycontent block**" in the **_base.htm** file – replace that block with the text in between the **block** and **endblock** template commands.

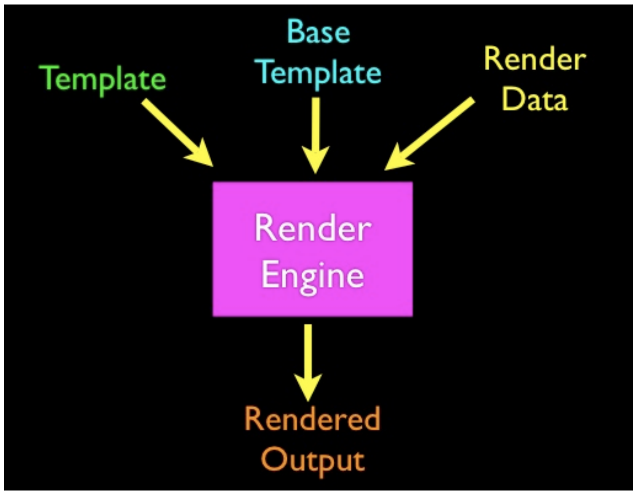
The **_base.htm** file is placed in the template directory along with all of the rest of the template files:



The contents of the **_base.htm** file are the common text we want to put into each page plus an indication of where the body content is to be placed:

```
<head>
   <title>App Engine - HTML</title>
   <link href="/static/glike.css" rel="stylesheet" type="text/css" />
 </head>
 <body>
   <div id="header">
      <h1><a href="index.htm" class="selected">
              App Engine</a></h1>
      <ul class="toolbar">
        <li><a href="sites.htm">Sites</a></li>
        <li><a href="topics.htm" >Topics</a></li>
      </ul>
   </div>
   <div id="bodycontent">
      {% block bodycontent %}
           Replace this
      {% endblock %}
   </div>
 </body>
</html>
```

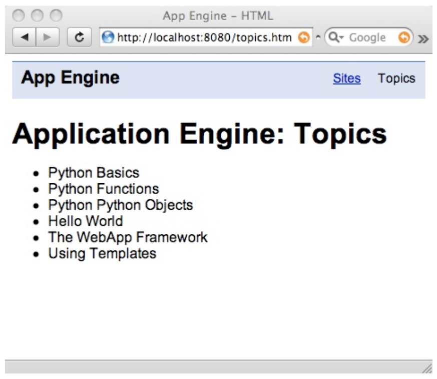
We include a template engine directive in the base template to indicate the beginning and end of the block that will be replaced by each template that uses (extends) **_base.htm**. The text "Replace this" will not appear in the resulting HTML after the render has been completed.



We do not need to make any change to the Controller code – the use of a base template is something that is completely handled in the **template.render()** call.

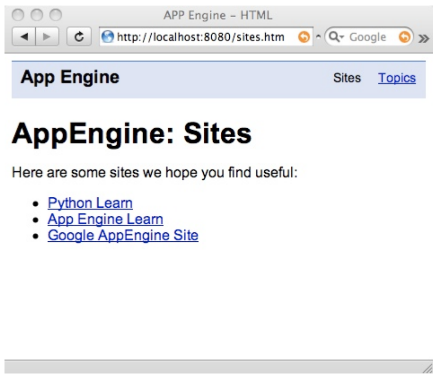
Conditional HTML using ifequal

Out application has several pages – and while we have moved most of the repeated text into a base file, there is one area in the **_base.htm** that needs to change between files. If we look at the pages, we see that as we move between pages, we want to have the navigation links colored differently to indicate which page we are currently looking at.



Application Engine: Topics

- Python Basics
- Python Functions
- Python Python Objects
- Hello World
- The WebApp Framework
- Using Templates



AppEngine: Sites

Here are some sites we hope you find useful:

- Python Learn
- App Engine Learn
- Google AppEngine Site

We make this change by using the selected class in the generated HTML. For example on the **topics.htm** file we need the "Topics" link to be indicated as "selected":

```
<ul class="toolbar">
  <li><a href="sites.htm">Sites</a></li>
  <li><a href="topics.htm" class="selected">Topics</a></li>
</ul>
```

We need to generate this text differently on each page and the generated text depends on the file we are rendering. The path indicates which page we are on – so when we are on the Topics page, the path is "/topics.htm".

We make a small change to the controller to pass in the current path in to the render process as follows:

```
def get(self):
    path = self.request.path
    try:
        temp = os.path.join(os.path.dirname(__file__), 'templates' + path)
        outstr = template.render(temp, { 'path': path })
        self.response.out.write(outstr)
    except:
        temp = os.path.join(os.path.dirname(__file__), 'templates/index.htm')
        outstr = template.render(temp, { 'path': path })
        self.response.out.write(outstr)
```

With this change, the template has access to the current path for the request. We then make the following change to the template:

```
<ul class="toolbar">
   <li><a href="sites.htm"
         {% ifequal path '/sites.htm' %}
               class="selected"
         {% endifequal %}
        >Sites</a></li>
   <li><a href="topics.htm"
         {% ifequal path '/topics.htm' %}
               class="selected"
```

```
            {% endifequal %}
          >Topics</a></li>
    </ul>
```

This initially looks a bit complicated. At a high level, all it is doing is adding the text class="selected" to the anchor (</a>) tag when the current path matches "/topic.htm" or "/sites.htm" respectively.

We are taking advantage of the fact that whitespace and end-lines do not matter in HTML. The generated code will look one of the following two ways:

```
        <li><a href="topics.htm"
                    class="selected"
            >Topics</a></li>
```

or

```
       <li><a href="topics.htm"
           >Topics</a></li>
```

While it looks a little choppy – it is valid HTML and our class="selected" appears when appropriate. Looking at the code in the template, we can examine the **ifequal** template directive:

```
            {% ifequal path '/topics.htm' %}
                  class="selected"
            {% endifequal %}
```

The **ifequal** directive compares the contents of the **path** variable with the string **"/topics.htm"** and conditionally includes the **class="select"** in the generated output.

The combination of the two **ifequal** directives means that the links give us the properly generated navigation HTML based on which page is being generated. This is quite nice because now the entire navigation can be included in the **_base.htm** file, making the page templates very clean and simple:

```
    {% extends "_base.htm" %}
    {% block bodycontent %}
          <h1>Application Engine: About</h1>
          <p>
          Welcome to the site dedicated to
          learning the Google Application Engine.
          We hope you find www.appenginelearn.com useful.
          </p>
    {% endblock %}
```

This approach makes it very simple to add a new page or make a change across all pages. In general when we can avoid repeating the same code over and over, our code is easier to maintain and modify.

More on Templates

This only scratched the surface of the template directives. The Google Template Engine comes from the Django project (www.django.org). You can read more about the template language features at:

http://docs.djangoproject.com/en/dev/ref/templates/builtins/?from=olddocs

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