Templates

- While we could write all of the HTML into the response using `self.response.out.write()`, we really prefer not to do this.
- Templates allow us to separately edit HTML files and leave little areas in those files where data from Python gets dropped in.
- Then when we want to display a view, we process the template to produce the HTTP Response.

http://docs.djangoproject.com/en/dev/ref/templates/builtins/?from=olddocs
class MainHandler(webapp.RequestHandler):
    formstring = """<form method="post" action="/">
    <p>Enter Guess:
    <input type="text" name="guess"></p>
    <p><input type="submit"></p>
    </form>"
    
def get(self):
        self.response.out.write("<p>Good luck!</p>\n")
        self.response.out.write(self.formstring)
    
def post(self):
        stguess = self.request.get('guess')
        logging.info('User guess=' + stguess)
        try:
            
            YUCK!!
            Python strings are a *lousy* way to store
            and edit HTML. Your code gets obtuse and
            nasty. Let's move the HTML into a separate
            file.

Separation of Concerns

- A well written App Engine Application has no HTML in
  the Python code - it processes the input data, talks to
  databases, makes lots of decisions, figures out what to
do next and then
- Grabs some HTML from a template - replacing a few
  selected values in the HTML from computed data - and
  viola! We have a response.

Terminology

- We name the three basic functions of an application as
  follows
- Controller - The Python 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
MVC

- We call this pattern the “Model - View - Controller” pattern (or MVC for short)
- It is a very common pattern in web applications - not just Google Application Engine
  - Ruby on Rails
  - Spring MVC
- We will meet the “Model” later - for now we will work with the View and Controller

Back to: Templates

- A template is mostly HTML but we have some little syntax embedded in the HTML to drop in bits of data at run-time
- The controller computes the “bits” and gives them to the “Render Engine” to put into the template.

A Simple Template

```html
<p>{{ hint }}</p>
<form method="post" action="/">
  <p>Enter Guess: <input type="text" name="guess"/></p>
  <input type="submit"/>
</form>
```

Mostly HTML - with a little place to drop in data from the Controller.
In The Controller

- In the controller, we prepare a Python Dictionary object with the data for the template and call the “Render Engine”

    outstr = template.render(filepath, { 'hint': 'Too low'})

The Render Engine takes two parameters (1) the path to a template file, and (2) a Python dictionary with key value pairs of the data areas in the template.

Template Pattern

- We store templates in a folder called “templates” under the main application directory to keep the templates (views) separate from the Python code (controller)

- We need to load the template from the right place in our Python code (it is a little ugly...)

    filepath = os.path.join(os.path.dirname(__file__), 'templates/index.htm')
    outstr = template.render(filepath, { 'hint': 'Too low'})
def post(self):
    stguess = self.request.get('guess')
guess = int(stguess)
if guess == 42:
    msg = 'Congratulations'
else:
    msg = 'Your guess is too low'
    msg = 'Your guess is too high'

outstr = template.render(temp, {'hint': msg, 'oldguess': stguess})
self.response.out.write(outstr)

We read the guess, convert it to an integer, check if it is right or wrong, setting a message variable and then passing some data into a template to be rendered.

Good luck!
Enter Guess: 25

Controller and View

<form method="post" action="/">
    <p>Enter Guess: <input type="text" name="guess"/></p>
    <p><input type="submit"></p>
</form>

Your Guess: 25
Your guess is too low

View

Template Summary

- We separate the logic of our program (Controller) from the HTML bits of the program (View) to keep things cleaner and more organization
- We use the Google templating engine to read the templates and substitute bits of computed data into the resulting HTML

Application Structure

- We keep the app.yaml and index.py files in the main application folder and the templates are stored in a folder called “templates”
- This is not a *rule* - just a pattern that it makes it easier to look at someone else's code
Several Templates
Program: ae-05-templates

Real Applications

- Real applications have lots of handlers and lots of templates
- In this section we start to look at techniques for managing and organizing templates

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

Our Application

Our Application has three pages - no forms, and a bit of CSS to make the navigation pretty and light blue. It is mostly a static site.

Application Layout

- There are three templates in the templates directory
- The CSS file is in the static directory - this is a special directory
Looking at app.yaml

- The app.yaml file has a new handler for static data which does not change like images, CSS, javascript libraries, etc
- Google serves these "read-only" files *very* efficiently
- Identifying them as static can save you money

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

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

Looking at app.yaml

- The handlers in the app.yaml file are checked in order
- First it looks at the url to see if it starts with "/static"
- The last URL is a catch-all - send everything to the controller (index.py)

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

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

Controller Code

- The controller code is going to be very general
- It will look at the path on the URL and try to find a template of that name - if that fails, render the index.htm template

http://localhost:8080/topics.htm

For this URL, the path is /topics.htm
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)

http://localhost:8080/topics.htm

If all else fails, render templates/index.htm
Note that we are *not* passing any data to the templates.

if __name__ == '__main__':
    app = webapp2.WSGIApplication([('/', MainHandler)], debug=True)
    webapp2.run(app)

http://localhost:8080/topics.htm

path = self.request.path
temp = os.path.join(...'templates' + path)
outstr = template.render(temp, { })
self.response.out.write(outstr)

The browser also does a GET request for /static/glike.css

In the Log....

Terminal — Python — 90×21

Extending Base Templates
Program: ae-06-templates

www.appenginelearn.com
Base Templates

- When building web sites there is a great deal of common material across pages
  - head
  - navigation
- Often only a small amount of information changes between pages

These files are nearly identical. And we have lots of files like this.

Application Layout

- This is the same as the previous application except we refactor the templates, putting the common material into the file _base.htm
- We reuse the _base.htm content in each of the other templates

A Base Template

- We create a base template that contains the material that is common across the pages and leave a little place in the base template to put in the bits that change
Welcome to the site dedicated to learning the Google Application Engine. We hope you find www.appenginelearn.com useful.
Extending a Base Template

- This capability to extend a base template is just part of the standard template render processing
- The template which is rendered is “index.htm”
- The render engine reads through index.htm. It sees the extend directive and goes to get the content of _base.htm as the starting point for index.htm

```html
{% extends "_base.htm" %}
{% block bodycontent %}
<h1>Application Engine: About</h1>
...
{% endblock %}
```

Making Navigation Look Nice

Program: ae-06-templates

www.appenginelearn.com

Navigation Issues

- As we navigate between pages, we want the look of the "current" page to change color or provide some indication which page we are on.
- This is usually done with a CSS class on the <li> tag

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

In topics.htm, the style sheet changes the Topics link to be Black and not underlined.

```html
a.selected {
  color: black;
  text-decoration: none;
}
```
Problem

- In this situation - the link that is selected changes between pages
- We need to put class="selected" on <a> tag for the current page but not for the other pages

Solution

- We pass the current path for the page into the template as a render parameter
- In the template we *check* the current path and only emit the class="selected" when the path is the current page

```python
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, { '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)
```

For each of the links, if the path matches, we emit class="selected" otherwise we do not.

**Conditional HTML generation.**

```html
<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>
```
Our Application

More on Templates

- This is only scratching the surface of templates
- The Google Application Engine templating language is taken from the django application
- You can read further in the django documentation

Summary

- We can use the ability to create a base template and then extend it in our regular templates to reduce the amount of repeated HTML code in templates.
- We can even make pretty navigation links which change based on which page is the current page
- When we don’t have to repeat the same code over and over - it is easy to make changes without breaking things