MIT AITI
Mobile, Python, Software Development

Django Blog Project
Part 4 – HTML and Templates

You will be adding templates to your blog app. These templates will allow you to view your blogs and comments. If you get stuck, take a look at these resources:

1. Lecture slides
2. Previous labs
3. Other group members
4. Django documentation
    a. Template Syntax overview
    b. Built-in tags and filters
5. Google
6. Instructors

To play around with HTML, there is an online HTML editor available at
http://www.w3schools.com/html/tryit.asp?filename=tryhtml_basic

Setting up your template directory

1. Change to the directory of the project (the same folder as settings.py) and create directories where you will store templates:
```
mkdir templates
mkdir templates/blog
```

2. Ok now open up your settings.py file. Because we are adding files that Django needs to find (templates) we need to add a setting:

```
SITE_ROOT = os.path.dirname(os.path.realpath(__file__))
```

This variable is the name of the folder in which settings.py is located, and will allow us to specify the location of other folders *relative* to the current one, rather than *absolutely*. This way, if you move your project to a new place, or a collaborator opens it on his or her computer, it will *just work*.

3. Edit TEMPLATE_DIRS in setting.py to be
```
TEMPLATE_DIRS = (
    os.path.join(SITE_ROOT,'templates'),
    …
)
```

This will tell Django to look in a folder called 'templates', in the same folder as your settings.py file, for the templates mentioned in your blog app.

4. Change directory to templates/blog (the easiest instruction yet!)
   `cd templates/blog`

Creating templates

5. Create a new template file "base.html" in "templates/blog".
   It should include:
   - A <head> section containing
     - <title> tags (title it whatever you want; I called mine 'Blog')
   - A <body> section containing
     - A link to '/blog/posts/' containing a heading with the title of your blog
     - A with id "content" that contains an empty django {% block %} called content. Don't forget to add the {% endblock %} tag

   Try to write this yourself – but if you get stuck there is an answer in the file 'instructions/part_4_instructor_code/answer_step_6.html"

6. Create a new template file "post_list.html" in templates/blog. This template should *extend* the base template – remember that you must identify the file with "base/blog.html", not just "blog.html".

   It should also override the {% block content %} that you defined in base.html in step 6 with html and template code that does the following:

   For each post in the variable "posts", there should be:

   - A subheading (think <h[23456]>) containing a link to the detail page of the post
     (NOTE: read through the file "instructions/linking_to_objects.pdf" to learn about the different ways of doing this)
   - An unordered list <ul> with three list items <li>:
     - Created: {{ the post created date }}
     - Last updated: {{ the date the post was last updated }}
     - {{ the first 60 characters of the body of the post (remember that one??) }}

   Try to do this all yourself, but if you can't, a solution to what should be inside of the content block is given in "instructions/part_4_instructor_code/answer_step_7.html"

7. Create a new template file "post_detail.html" in "templates/blog". The purpose of post_detail.html will be to display a single post and all of its comments. Edit post_detail.html so that it extends base.html and overrides the content block in order to display

    - A post's title, body, date of creation, and date of last update. Assume that you have access to the relevant Post object with the variable named `post`. You can inspect the post with `{{ post.title }}`, `{{ post.body }}`, etc.
    - All comments associated with the post. For each comment, show the comment's author, body, date of creation, and date of last update. Assume that you have access to a variable `comments` which is a list of all Comment objects associated with the blog. Each Comment object `c` in `comments` can be inspected with `{{ c.author }}`, `{{ c.body }}`, etc. *Hint: do a for loop over `comments` using the {% for … %} tag.*

    Sorry… no answers for this one! Use the past two steps as a guide to help you on this one.

8. Create a new template post_search.html in templates/blog. The purpose of post_search.html is to list all posts that have a match against a search query. Edit post_search.html so that it extends base.html and overrides the content block in order to display

    - The search query term. Assume that it is stored in the variable `term`.
    - A list of the titles of all posts that matched the query. Assume that you have a list of the Post objects that matched in the variable `posts`. Each title may also be a link to the post itself.

Directing URLs to templates

All the hard parts are done – you have made the models, views, and templates. In the last part of this lab you connected URLs to views and views to models – and today is the last step: connecting views to templates.

9. Now that you've created templates, the next step is to declare when these templates are to be evaluated. You will do this by calling templates in the functions in views.py. Since these functions are called in response to regex matches in urls.py, you will effectively be mapping URLs to templates.

   Make sure that views.py has the following import statement

   ```
   from django.template import Context, loader
   from django.shortcuts import render_to_response
   ```

10. Edit your post_list function in views.py to look like

    ```
    def post_list(request):
        posts = Post.objects.all()
        t = loader.get_template('blog/post_list.html')
        c = Context({'posts':posts })
        return HttpResponse(t.render(c))
    ```

    The first line `posts = Post.objects.all()` is the same from the last part, but the last three lines use new functions that you haven't seen before (except in the slides I guess)

    - `loader.get_template('blog/post_list.html')` loads the post_list.html template, reading it and preparing it to be turned into a webpage
    - `Context({'posts': posts })` maps the name 'posts' to the actual variable posts. This mapping will be used to find the posts variable from within the posts_list.html template. In general, the Context function takes a dictionary where the keys are names of variables to be used in the loaded template and the values are the python variables to which you want them to refer. This mapping is called a context.
    - `return HttpResponse(t.render(c))` simply returns the HTML found by evaluating the template and filling in the values you specify with the values in the context.

11. Go to http://localhost:8000/blog/posts to see a list of your blog posts!

12. Edit the home function in views.py to look like

```
def home(request):
    return render_to_response('blog/base.html',{})
```

You may note that this function is a lot shorter than the function above. That is because of the shortcut `render_to_response!` It takes as arguments the name of a template and a dictionary and does four things:

1. **Loads** the template given as the first argument (using `loader.get_template`)
2. Creates a **Context** using the dictionary passed as the second argument as the mapping
3. **Renders** the loaded template from step 1 with the Context from step 2.
4. **Returns** an **HttpResponse** that is ready to be returned to the visitor.

This shortcut saves a lot of work and reduces the possibility of mistakes. Use it as often as possible rather than manually loading a template, creating a context, rendering the template, and creating an HttpResponse to return to the user.

13. Go to http://localhost:8000/blog/ to see the home page of your blog. The page should simply contain a link to the list of your blog posts.

14. Edit your post_detail function in views.py so that it loads and renders the template post_detail.html and uses the mapping `{'post': post, 'comments':comments)` as the context to render the template.

15. Go to http://localhost:8000/blog/posts and then click on one of your posts to see its details and comments.

16. Edit your post_search function in views.py to load and render the template "post_search.html" and uses the context {'posts' : posts, 'term':term} to render_and_return the template.

17. Let <term> be a word in the title of one of your blog posts. Go to http://localhost:8000/blog/posts/search/<term> to see a list of posts that match a search for the term <term>.

18. WHOOHOO!!!! YOU HAVE MADE A WEBSITE IN DJANGO. **PUSH IT TO GITHUB AND HEROKU.** Then Start dancing. If you are unhappy with how your pages look (which is probably the case – the default html styling is ugly – check out CSS and styling. For a quick way to get your pages looking **a lot** better, google "Twitter Bootstrap" and learn how to use it.)