Sessions, Templates, Forms

Maxim Mozgovoy

Sessions

- Sometimes you need to save some data *between* the requests.
- E.g., if you need to remember that a certain user is logged in.
- This is achieved with *sessions*:
  - the data to be saved is stored on the server;
  - it is retrieved with a key stored in a cookie on a user device.
- Django has built-in mechanism for user authentication, but we will implement a simple method ourselves as an exercise.

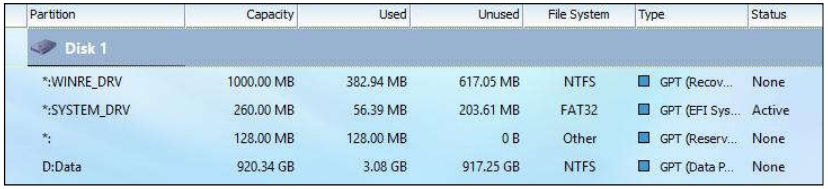
Sessions

Using sessions in Django is very easy:

- Each `HttpRequest` object has a `session` attribute.
- It is a dictionary of session-stored values.
- We can keep any number of values associated with a text key.
- Use `get(key, default=None)` to check for potentially non-existent keys:
  `r = request.sesson.get('user_known', False)`
- Set values with ordinary dictionary syntax:
  `request.session['user_known'] = True`

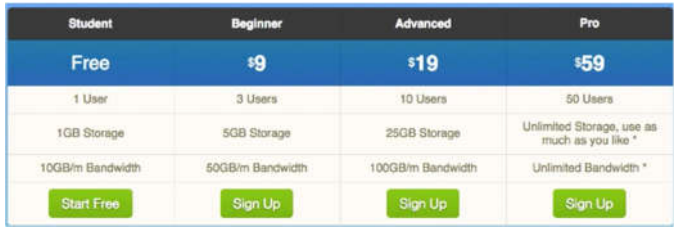
Templates in Django

- Separation of design and content is one of serious issues in web programming. Compare:



Partition	Capacity	Used	Unused	File System	Type	Status
Disk 1						
*:WINRE_DRV	1000.00 MB	382.94 MB	617.05 MB	NTFS	GPT (Recov...	None
*:SYSTEM_DRV	260.00 MB	56.39 MB	203.61 MB	FAT32	GPT (EFI Sys...	Active
*:	128.00 MB	128.00 MB	0 B	Other	GPT (Reserv...	None
D:Data	920.34 GB	3.08 GB	917.25 GB	NTFS	GPT (Data P...	None

- In desktop apps, the design of user controls is strictly separated from code and content.



Student	Beginner	Advanced	Pro
Free	$9	$19	$59
1 User	3 Users	10 Users	50 Users
1GB Storage	5GB Storage	25GB Storage	Unlimited Storage, use as much as you like *
10GB/m Bandwidth	50GB/m Bandwidth	100GB/m Bandwidth	Unlimited Bandwidth *
Start Free	Sign Up	Sign Up	Sign Up

- In web apps, the design of user controls is specified inside HTML along with text and JavaScript code.

Templates in Django

- Complete separation of code / design / content is not possible, but most frameworks are trying to do their best.
- Still, some knowledge of HTML is necessary.
- Django separates HTML design from app logic with templates.
- Each template is simply an HTML document with special placeholders that can be substituted with Django objects converted to HTML.

Trivial Template Example

Trivial example: just open the file `index.html` and return it as a result of an HTTP request.

```
from django.http import HttpResponse
from django.template import loader

def index(request):
    template = loader.get_template('index.html')
    return HttpResponse(template.render())
```

By default, template files are being searched inside the application folder, but you can adjust this behavior.

Passing Variables to HTML

Often we need to pass variables to the resulting HTML.
Without templates, we have to use simple Python string functions:

```
usr_name = "John"
usr_car = "Toyota"
r = "{} has a car {}.".format(usr_name, usr_car)

return HttpResponse(r)
```


Passing Variables to HTML

With templates, we can separate design from data:

```
<!-- HTML code (usercar.html) -->
User <b>{{ user }}</b> has a car <b>{{ car }}</b>

# Python code
def myview(request):
  usr_name = "John"
  usr_car = "Toyota"
  c = {'user': usr_name, 'car': car_name}
  tpl = loader.get_template('usercar.html')

  return HttpResponse(tpl.render(c, request))
```

Using render() Shortcut

Utility `render()` function makes this code a bit shorter:

```
<!-- HTML code (usercar.html) -->
User <b>{{ user }}</b> has a car <b>{{ car }}</b>

# Python code
from django.shortcuts import render

def myview(request):
  usr_name = "John"
  usr_car = "Toyota"
  c = {'user': usr_name, 'car': car_name}

  return render(request, 'usercar.html', c)
```


Class Variables in Templates

Templates support class variables:

```
<!-- HTML code (usercar.html) -->
User <b>{{ user.name }}</b> has
a car <b>{{ user.car }}</b>

# Python code
class User:
  def __init__(self, name, car):
    self.name = name
    self.car = car
…

c = {'user': User("John", "Toyota")}
return render(request, 'usercar.html', c)
```


Handy Tags: for…endfor

Templates may contain special tags inside `{%...%}` sequences. They control HTML generation.

Tag `for`: duplicates fragments of HTML for different values taken from the input list:

```
c = {'people': ['John', 'Paul', 'Nick']} # Python
…

<!-- HTML -->
<ul>
{% for p in people %} <li>{{ p }}</li>
{% endfor %}
</ul>
```


Handy Tags: for…endfor

```
<!-- HTML -->
<ul>
{% for p in people %} <li>{{ p }}</li>
{% endfor %}
</ul>
```

Becomes after substitution with `render()` function:

```
<ul>
<li>John</li>
<li>Paul</li>
<li>Nick</li>
</ul>
```

Handy Tags: cycle

The tag `cycle` is substituted with its next argument in a loop:

```
<ul>
{% for p in people %}
<li><font color={% cycle 'red' 'black' %}>
{{ p }}</font></li>
{% endfor %}
</ul>
```

This fragment becomes:

```
<ul>
<li><font color=red>John</font></li>
<li><font color=black>Paul</font></li>
<li><font color=red>Nick</font></li>
</ul>
```


Handy Tags: if…elif…else…endif

The tag `if` is used to conditionally output or skip the block of text:

```
{% if count == 1 %}
One
{% elif count == 2 %}
Two
{% else %}
Many
{% endif %}
```

This tag is compatible with Boolean operators `and`, `or`, `not`, and operations like ==, !=, <, >, etc.

HTML Forms

- HTML forms are used to group user controls that should be processed in one request.
- For example, “login” and “password” text fields and “login” button typically belong to the same form.
- A form is described within <form>…</form> tags in HTML.
- When creating a form, one must specify the target URL and the method used to send data:

  `<form action=`*`URL`* `method=`*`METHOD`*`>`

- The URL should point to the page that process the request.

GET vs POST

- The method used to send data should be either GET or POST.
- GET: requests data from the page;
  POST: submits data to the page.
- GET encodes data in the URL:
  https://www.wolframalpha.com/input/?i=mass+of+moon
- As a rule, you should select GET if:
  - You want the users to be able to bookmark the URL;
  - You want the browser to be able to cache the page;
  - You don’t deal with sensitive data (such as passwords).
- POST method should be used to:
  - Submit sensitive data (passwords).
  - Send large amount of data (so it can’t be encoded in a URL).
  - Process requests that shouldn’t be cached or remain in history.

Django Forms

- You can design user-input forms in HTML and use Django template mechanism to display them.
- However, it might be more convenient to design forms using Django Python library.
- This method also makes easier processing user-supplied data.
- Django form objects can be used to construct everything *inside* the form, except <form>, </form> tags and the submit button.
- A typical HTML snippet for a form object looks like this:

```
<!-- username.html -->
<form action="process_user" method="post">
 {% csrf_token %} <!-- needed for POST -->
 {{ user_form }}
 <input type="submit" value="Submit" />
</form>
```


Django Forms: Simple Example

Let's design a simple form that implements user name input:

```
from django import forms

class UserNameForm(forms.Form):
  firstname = forms.CharField(label='First name',
                              max_length=20)
  lastname = forms.CharField(label='Last name',
                             max_length=20)
```

Note: forms are typically defined in a separate file (you can name it forms.py, for example)

Django Forms: Simple Example

It can be used in views as follows:

```
def fill_user_form(request):
    f = UserNameForm() # create an empty form
    return render(request, 'username.html',
                           {'user_form': f})

def process_user(request):
    f = UserNameForm(request.POST)
    ... # process form data
```

Note that you can directly insert user-provided data into the database, since Django prevents SQL injection attacks!

Form Fields

Some useful form elements ("fields"):

```
BooleanField     -- a True/False checkbox
CharField        -- a textbox
ChoiceField      -- a drop-down list with choices
DateField        -- inputs date
DateTimeField    -- inputs date and time
EmailField       -- an email address
FileField        -- a file upload box
FloatField       -- a floating-point number
ImageField       -- an image upload control
IntegerField     -- an integer number
```

Each field is associated with a *widget* that works as a corresponding HTML element. You can change default widgets:

```
p = forms.CharField(label='Pass', max_length=10,
                    widget=forms.PasswordInput())
```

Form Validation

- Each form field has constructor arguments that define a set of valid values for the field. Check them here: https://docs.djangoproject.com/en/2.2/ref/forms/fields/
- A very important ability of a field object is to tell whether it contains a valid content (via `is_valid()` method).
- The whole form can be checked with a single `is_valid()` call of the form object:

```
f = UserNameForm(request.POST)
if f.is_valid(): # process form data
  firstname = f.cleaned_data['firstname']
  lastname = f.cleaned_data['lastname']
  ...
else:
  errors = f.errors # get errors as a dictionary
```


Login/password management

- User passwords should not be kept as a plain text.
- After we requested a user password, we should encrypt it and store in this encrypted form.
- To check whether a certain password is correct, we encrypt it, too, and compare encrypted versions.
- Django provides two convenience functions for this process:

```
from django.contrib.auth.hashers import
                make_password, check_password
enc_pass = make_password('MyPa$$_256!!')

# returns True if the password matches
r = check_password('MyPa$$_256!!', enc_pass)
```


Dynamic Form Setup

Sometimes you need to adjust form fields after the form is constructed. This can be done by accessing them with `self.fields['field-name']` syntax:

```
class MyForm(forms.Form):
  cities = forms.ChoiceField(label='City')

  def set_cities(self, c_list):
    self.fields['cities'] = c_list


...
f = MyForm()
f.set_cities([(1, 'Tokyo'), (2, 'Paris')])
```


Coming Exercise

It is difficult to work on views without any user interface.

Let's implement some basic interface for our application.

Use the simplest GUI possible: now our task is to make the software work properly, and we can make the UI pretty later.

Also now we can finally implement user authentication.