Open Source Frameworks for Rapid Application Development

Marek Krętowski
Krzysztof Bandurski, Tomasz Łukaszuk, Tomasz Rybak

Software Departament
Faculty of Computer Science
Bialystok University of Technology

m.kretowski@pb.edu.pl
k.bandurski@pb.edu.pl, t.lukaszuk@pb.edu.pl, t.rybak@pb.edu.pl

Lecture topic

Django templates

Django templates: Table of content

General

What are templates?

- A template is simply a text file that the template engine can use to generate ANY text-based format (HTML, XML, CSV, JavaScript etc.)
- `{{ variables }}` are passed to the template as a normal dictionary of `variable : value` pairs and get replaced with values when the template is evaluated
- `{{ variables }}` can be modified by filters as follows: `{{ variable|filter }}`
- `{% tags %}` control the logic of the template
- `{% tags %}` function similarly to programming constructs...
- ...but they are NOT simply Python embedded into HTML.
- Philosophy: the template system is meant to express presentation, not program logic

Example

Note the use of `{{ variables }}`, `{% tags %}` and `{{ variable|filter }}` combinations

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

Basic use in views

```
from django.shortcuts import render_to_response

def my_view(request):
    # View code here...
    return render_to_response('myapp/index.html', {"foo": "bar"},
        mimetype="application/xhtml+xml")
```

`render_to_response()` is a helper function. Without it, we would have to write:

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

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/template.html')
    c = Context({'foo': 'bar'})
    return HttpResponse(t.render(c),
        mimetype="application/xhtml+xml")
```

Note that the path to the template is relative

Loading templates

To load a template, Django uses template loaders in the order that they appear in the `TEMPLATE_LOADERS` setting:

```
('django.template.loaders.filesystem.load_template_source',
 'django.template.loaders.app_directories.load_template_source')
```

1. `loaders.filesystem.load_template_source` looks into directories specified in the `TEMPLATE_DIRS` setting
2. `loaders.app_directories.load_template_source` looks into the “templates” directory of each intalled app.

Templates for designers

More on variables

- When the template engine encounters a variable, it evaluates that variable and replaces it with the result.
- Use a dot to access attributes of a variable (e.g. `{{ section.title }}`). The following lookups will be tried, in this order:
  1. Dictionary lookup
  2. Attribute lookup
  3. Method call
  4. List-index lookup
- If a variable doesn't exist, the stystem will insert the value of the `TEMPLATE_STRING_IF_INVALID` setting (empty string by default)

More on filters

- Filters are used to modify variables
- Filters can be chained,
  e.g. `{{ text|escape|linebreaks }}`
- Some filters can take arguments, e.g. `{{ list|join:", " }}` (only arguments with spaces must be quoted)
- Common filters:

```
{{ value|default:"nothing" }}
```

if `value` is missing or empty, display “nothing”

```
{{ value|length }}
```

if `value` is “cat”, the output will be 3

```
{{ value|striptags }}
```

if `value` is "Joel <button>is</button> a slug", the output will be “Joel is a slug”

More on tags

- Some tags create text, some control flow by performing loops and logic, some load external information into the template
- Some tags require beginning and ending tags:

  `{% tag %} ... tag contents ... {% endtag %}`
- Common tags:

```
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
```

Loops over each item in a sequence.

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% else %}
    No athletes.
{% endif %}
```

Evaluates variable and displays its contents if the variable is “True”

Template inheritance

- The most powerful and complex part of the template engine
- Allows to build a base “skeleton” template that contains all the common elements of your site and defines blocks that child templates can override
- Extend templates using the `{% extends %}` tag, e.g.
  `{% extends "base.html" %}`
- Define blocks using the `{% block %}` tag, e.g.
  `{% block content %}{% endblock %}`

Inheritance example

base.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>
<body><div id="content">
        {% block content %}{% endblock %}
</body></div>
```

child template:

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}
{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
{% endfor %}
{% endblock %}
```

Tips on iheritance

- If used, `{% extends %}` must be the first template tag in the template
- Child templates do not have to define all parents blocks.
- If you duplicate content in many templates, you should move that content to `{% block %}` in a parent template
- Get the content from the block from the parent template using the `{{ block.super }}` variable.
- For extra readibility, give a *name* to `{% endblock %}`, like this:

```
{% block content %}
...
{% endblock content %}
```

Automatic HTML escaping

Let's consider this template fragment:

```
Hello, {{ name }}.
```

If a user enters his name as:

```
<script>alert('hello')</script>
```

the template will be rendered as:

```
Hello, <script>alert('hello')</script>
```

and the browser would pop-up a JavaScript alert box!

- User-submitted data cannot be trusted
- By default, Django escapes the output of every variable tag ("<" is converted to "&lt" and so on)
- This behaviour can be turned off for a variable or a block.

Custom tag and filter libraries

The `{% load %}` tag allows to access custom tags and filters:

```
{% load comments %}

{% comment_form for blogs.entries entry.id with is_public yes %}
```

- `{% load %}` can take multiple library names at once
- It affects only the current template

Templates for programmers

How do templates work?

Using the template system in Python is a two-step process:

1. Compile the raw template code into a Template object.

```
>>> from django.template import Template
>>> t = Template("My name is {{ my_name }}.")
>>> print t
<django.template.Template instance>
```

2. Call the `render()` method of the Template object with a given context.

```
>>> c = Context({"my_name": "Adrian"})
>>> t.render(c)
"My name is Adrian."

>>> c = Context({"my_name": "Dolores"})
>>> t.render(c)
"My name is Dolores."
```

Context Processors and `RequestContext`

`RequestContext` takes `request` as the first argument:

```
c = RequestContext(request, {
    'foo': 'bar',
})
```

It populates the context with variables provided by context processors defined in the `TEMPLATE_CONTEXT_PROCESSORS` setting that defaults to:

```
("django.core.context_processors.auth", # user, messages, perms
"django.core.context_processors.debug", # debug, sql_queries
"django.core.context_processors.i18n",  # LANGUAGES, LANGUAGE_CODE
"django.core.context_processors.media") # MEDIA_URL
```

- Context processors populate the context with data based on `request`
- Context processors will override variables supplied by the user, if their names overlap.

Writing custom tag and filter libraries

Custom tags and filters should live in the “templatetags” directory of an appliation, e.g.:

```
polls/
    models.py
    templatetags/
        __init__.py
        poll_extras.py
    views.py
```

To be a valid tag library, poll_extras.py must have the following variable:

```
from django import template

register = template.Library()
```

This variable will be used to register all custom tags and filters.

Writing custom filters

Custom filters are just Python functions that take one or two arguments:

- The value of the variable (input) – not necessarily a string.
- The value of the argument – this can have a default value, or be left out altogether.

Examples:

```
def cut(value, arg):
    "Removes all values of arg from the given string"
    return value.replace(arg, '')
```

```
def lower(value): # Only one argument.
    "Converts a string into all lowercase"
    return value.lower()
```

Registering filters:

```
register.filter('cut', cut)
register.filter('lower', lower)
```

Writing custom tags

- Tags are far more complex than filters, because they can do anything.
- Writing a custom tag from scratch is complicated, and so is explaining it
- Luckily, Django provides helper functions to write commonly used types of tags: simple tags and inclusion tags

Simple tags

Simple tags take a number of arguments and return a string after doing some processing. Example: a tag that returns the current time, formatted according to the argument given:

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

This tag could be written as:

```
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)

register.simple_tag(current_time)
```

or, using Python 2.4 decorated syntax:

```
@register.simple_tag
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)
```

Inclusion tags

Inclusion tags display some data by rendering another template.
Example: a tag that displays the results of a poll:

```
{% show_results poll %}
```

This tag could be written as:

```
def show_results(poll):
    choices = poll.choice_set.all()
    return {'choices': choices}
```

and registered as:

```
register.inclusion_tag('results.html')(show_results)
```

results.html:

```
<ul>
{% for choice in choices %}
    <li> {{ choice }} </li>
{% endfor %}
</ul>
```

References

- Django documentation: http://www.djangoproject.com/
- Djangobook: http://www.djangobook.com/