Pyroutes Documentation

Release 0.5.0

Kristian Klette, Morten Minde Neergaard, Magnus Eide, Dalton Ma

November 25, 2012

CONTENTS

pyroutes is a really small framework (more of a wrapper really) around wsgi for developing small python web apps. If you're developing a larger project, I suggest you point your browser to http://djangoproject.com instead :-)

CHAPTER

ONE

RELEASE CHANGELOGS

1.1 Changelog

1.1.1 Release 0.5.0

- Fixed Cookie path default value to be relative to base site location.
- Added AppendSlashes middleware for forcing URLs to end in slash. Not on as default.
- Changed handling of requests against invalig URLs (We now process all requests as if their URLs had a leading slash, regardless...)
- Created `Responsify` middleware, and included it as default. This allows you to return from a route withoute wrapping your response in a Response object.
- Ensured compatability across Python versions 2.4 through 3.2

1.1.2 Release 0.4.1

- Rename `find_request_handler` to `find_route`.
- Rename some internal variables, notably all references to `handler` that were actually a `Route` instance have been renamed to route.
- Minor fixes.

1.1.3 Release 0.4.0

- Implemented redirection relative to Application's root. See issue #1 on github: https://github.com/pyroutes/pyroutes/issues/1
- Fixed argument defaults being overwritten. See issue #5 on github: https://github.com/pyroutes/pyroutes/issues/3
- Altered logic for matching routes (stricter matching of argument count)
- Code simplification on many minor details
- Some minor regressions introduced by the middleware fixed

1.1.4 Release 0.3.1

- Remove leftover print statement causing errors using mod_wsgi

1.1.5 Release 0.3.0

- Added support for autopopulating handler method with data from the URL.
- Added middleware support
- New and improved documentation
- Lots of cleanups on the code.

1.1.6 Release 0.2.2

- Fixed bug where setting the TEMPLATE_DIR-option in `pyroutes_settings.py` would cause the default 404,403 and 400 error pages to not work.
- Fix bug where pyroutes would add two content-type headers to responses. (Thanks to Dalton Barreto)
- Fixed IF_MODIFIED_SINCE handling in utils.fileserver on windows.

1.1.7 Release 0.2.1

- Reduce setup.py dependencies to only distutils.
- Fix packaging of default templates
- Fix pypi-package complaining about README file missing.
- Fix unstable cookie handling in some cornercases.

1.1.8 Release 0.2.0

- New Request object included to every route. *Backward incompatible*
- New cookie handling framework
- Automatic HTTP-status code lookups in Response-objects.
- Project settings
- Better debug-pages when DEBUG=True in settings.
- Development fileserver
- Development autoreloader

CHAPTER

TWO

USER MANUAL

Contents:

2.1 Installation

pyroutes is available trough pypi and is easily installed by using either pip or easy_install.

2.1.1 Installation with pip

```
# pip install pyroutes
```

2.1.2 Installation with easy_install

```
# easy_install pyroutes
```

2.1.3 Following the development

You can follow the development version by checking out the source code from GitHub.

WARNING: There is no guarantee using the development branch won't break your projects. You've been warned.

```
$ git clone git://github.com/klette/pyroutes.git
$ cd pyroutes
$ sudo python setup.py install
```

2.2 Let's do it! (aka Quickstart)

All the following examples are also available in the `examples` folder of pyroutes, as `quickstart.py`.

2.2.1 Application entry point

The application entry point is located directly in the `pyroutes` module. Just do

```
from pyroutes import application
```

in the file you want as a handler for `mod_wsgi` or your preferred deployment method. Let's call it `handler.py` for now.

2.2.2 Adding routes

Routes are the way for defining which methods should handle requests to which paths.

This is the most basic example:

```
from pyroutes import route

@route('/')
def index(request):
    return 'Hello world!'
```

Here we define that our index method should handle all requests to `/`, and return the famous «Hello world!» to the user.

We can add more routes:

```
@route('/sayhello')
def sayhello(request, name='world'):
    return 'Hello %s!' % name
```

... Easy as pie! Save these at the end of our `handler.py` file.

Route handling gotchas

After adding the two example routes, we have a handler for `/` and `/sayhello`. If you try to access `/foo` you will get an 404 exception. However, accessing `/sayhello/master` does something quite different :)

2.2.3 Starting the development server

Pyroutes includes a development server to ease local development quite a bit. It's located in the `pyroutes.utils` module.

Using it is as easy as adding this to `handler.py` from the previous examples.:

```
if __name__ == '__main__':
    from pyroutes.utils import devserver
    devserver(application)
```

Serving static media

Static media is normally served directly from the web server, but we need static files when developing locally as well. You can serve static files through the devserver using the `fileserver` route in `pyroutes.utils`. Change the previous code to something like this:

```
if __name__ == '__main__':
    from pyroutes import utils
    route('/media')(utils.fileserver)
    utils.devserver(application)
```

This will now serve anything you have in the folder called `media` in your working directory in the `/media` path. This behaviour can be modified in `pyroutes_settings`.

2.2.4 Firing it up

Let's try what we have so far. Open up a terminal, go to the directory where you saved the `handler.py` file, and execute it:

```
$ python handler.py
Starting server on 0.0.0.0 port 8001...
```

Your application should now be running on port 8001. Let's try it.:

```
$ echo `wget -q -O - http://localhost:8001/`
Hello world!
$ echo `wget -q -O - http://localhost:8001/sayhello/Pyroutes`
Hello Pyroutes!
```

2.2.5 Debugging

If something in the code was not correct, and an exception was thrown, you'll get a error page with not much information. You can see more about what went wrong if you enable pyroutes' debugging.

This is done by creating a file called `pyroutes_settings.py` in your `PYTHONPATH`. Create this file and add:

```
DEBUG=True
```

Now refresh the page with the error, and you'll get a lot more information to work with.

2.2.6 Using URLs as data

As of Pyroutes >= 0.3.0 using URLs as data for your handler really simple. Let's create an `archive` route as an example:

```
@route('/archive')
def archive(request, year, month=None, day=None):
    return 'Year: %s  Month: %s  Day: %s' % (year, month, day)
```

And let's try it:

```
$ echo `wget -q -O - http://localhost:8001/archive`
(This returns Http404 because year is an obligatory parameter)
$ echo `wget -q -O - http://localhost:8001/archive/2010`
Year: 2010  Month: None  Day: None
$ echo `wget -q -O - http://localhost:8001/archive/2010/02`
Year: 2010  Month: 02  Day: None
$ echo `wget -q -O - http://localhost:8001/archive/2010/02/03`
Year: 2010  Month: 02  Day: 03
$ echo `wget -q -O - http://localhost:8001/archive/2010/02/03/foobar`
(This returns HTTP 404 because archive only accepts four parameters)
```

This example should make the URL matching logic clear. Note: If a method accepts a referenced argument list in the from *args, it will match any subpath of its route address.

An example:

```
@route('/pathprint')
def archive(request, *args):
    return 'User requested /%s under /pathprint' % '/'.join(args)
```

2.2.7 Accessing request data

One common operation in developing web applications is doing stuff with user data. Pyroutes gives you easy access to the POST, GET and FILES posted to your request handler.

```
@route('/newpost')
def new_post(request):
    if 'image' in request.FILES:
        # Do stuff with image
        filename, data = request.FILES['image']
        data = data.read()
    category = request.GET.get('category','default')
    title = request.POST.get('title', None)
    if not title:
        return 'No title!'
    return 'OK'
```

Note: If multiple fields have the same name, the value in the respective dicts are a list of the given values.

2.2.8 Sending responses to the user

Every route must return an instance of `pyroutes.http.response.Response`, or one of it's subclasses. The former defaults to sending a `text/html` response with status code `200 OK`. Any data returned that wasn't wrapped in a Response object will also have these defaults applied (by the Responsify middleware)

We have the follow built-in responses:

```
Response(content=None, headers=None, status_code='200 OK',
        default_content_header=True)

Redirect(location)
```

content may be any string or iterable. This means you can do something like this:

```
@route('/pdf')
def pdf(request):
    return Response(open('mypdf.pdf'), [('Content-Type', 'application/pdf')])
```

Also available for convenience is the HttpException subclasses, also found under `pyroutes.http.response`. An example (assuming a method `decrypt` that can decrypt files by some algorithm):

```
@route('/decrypt_file')
def decrypt(request, filename, key):
    full_filename = os.path.join('secrets_folder', filename)
    if not os.path.exists(full_filename):
        raise Http404({'#details': 'No such file "%s"' % filename})
    try:
        return decrypt(full_filename, key)
    except KeyError:
        raise Http403({'#details': 'Key did not match file'})
```

2.2.9 C is for cookie..

Cookies are the de-facto way of storing data on the clients. Pyroutes uses secure cookies by default. This means that if a user edits his own cookies, pyroutes will not accept them. This is done by storing a HMAC-signature, based on the cookie its signing and the `SECRET_KEY` in your settings, along with the actual cookie.

Settings cookies:

```
@route('/cookie-set')
def set_cookies(request, message='Hi!'):
    response = Response('Cookies set!')
    response.cookies.add_cookie('logged_in', 'true')
    # Insecure cookie setting
    response.cookies.add_unsigned_cookie('message', message)
    return response
```

Retrieving cookies:

```
@route('/cookie-get')
def get_cookies(request):
    logged_in = request.COOKIES.get_cookie('logged_in')
    message = request.COOKIES.get_unsigned_cookie('message')
    if logged_in:
        return message
    raise Http403({'#details': 'Go away!'})
```

Deleting cookies:

```
@route('/cookie-del')
def get_cookies(request):
    response = Response('Cookies deleted!')
    response.cookies.del_cookie('logged_in')
    response.cookies.del_cookie('message')
    return response
```

2.2.10 Let's go templates!

Pyroutes bundles XML-Template, a template system created by Steinar H. Gunderson, which might seem a bit «chunky», but it really fast, and guarantees it's output to be valid XML (or in our case XHTML). The big difference between XML-template and most other template systems out there, is that XML-template is purely a representation layer. You don't have any logic in your templates.

Now, pyroutes has a small wrapper around XML-Template for handling the most common template task; having a base-template, and a separate template for your current task.:

```
from pyroutes.template import TemplateRenderer

tmpl = TemplateRenderer('base.xml')

@route('/')
def index(request):
    return tmpl.render('index.xml', {})
```

For more information about XML-Template, see *Introduction to XML::Template*.

2.3 Deployment

As Pyroutes is a WSGI-based framework, we have several methods of deploying our applications. The most common is to use mod_wsgi and Apache.

2.3.1 Apache2.2 and mod_wsgi

As this web framework is based on WSGI, we'll use mod_wsgi as our way of deploying our project behind the Apache webserver.

Let's start by setting up our base requirements (assuming you have pyroutes installed).

Debian based systems:

```
sudo aptitude install apache2 libapache2-modwsgi
sudo a2enmod wsgi
sudo /etc/init.d/apache2 reload
```

Other

- Install apache from www.apache.com
- Install mod_wsgi from http://code.google.com/p/modwsgi
- Enable mod_wsgi and restart apache (might be different on your platform):

  ```
  sudo a2enmod wsgi
  sudo apache2ctl restart
  ```

Once you have that installed we're going to need a `VirtualHost` for our project. Use this configuration as an example. More configuration options are available at mod_wsgi's website.:

```
<VirtualHost *>
        ServerName example.pyroutes.com
        ServerAdmin klette@pyroutes.com
        DocumentRoot /home/klette/dev/myproject/webroot

        WSGIScriptAlias / /home/klette/dev/myproject/handler.py
</VirtualHost>
```

The most important line here is the `WSGIScriptAlias` line. In the example we declare that every path under `/` should be handled by that python file. When using `pyroutes` this is the file where you do:

```
from pyroutes import application
```

and import all files and modules declaring routes.

If `WSGIScriptAlias` is set to a different path, e.g. `/projects/wiki`, Redirect responses starting with a slash are made relative to this path.

That should be about it, and your project should be running smoothly behind apache.

2.4 Middleware

Middleware is run around each request processing.

In Python we can display this as:

```
Outer(Middle(Inner(final_method)))(my param)
```

Each layer calls the next with the the same parameters as it was called with, and returns the same. This allows us to edit requests and responses globally with ease. Each layer of middleware can return e.g an error page without calling the next layer, so `final_method` is not guaranteed to be called.

2.4.1 Adding middleware

Adding middleware is done by modifying the project settings. See *Settings* for more information about this.

2.4.2 Creating your own

Creating your own middleware is quite easy. Let's create a simple logging middleware as an example.

```
import logging

class LoggingMiddleware(object):
    def __init__(self, passthrough):
    self.logger = logging.getLogger()
    self.logger.addHandler(logging.FileHandler('/tmp/mylog.txt'))
    self.logger.setLevel(logging.DEBUG)
    self.passthrough = passthrough

def __call__(self, request):
    self.logger.debug('Got request %s', request)
    response = self.passthrough(request)
    self.logger.debug('Got response %s', response)
    return response
```

So, what's going on here?

Every middleware must accept a `passthrough` parameter. This is the next method in the middleware chain (or the handler itself). In the `__call__` method we accept a mandatory `request` parameter, and return the result of the `passthrough`-method, called with `request` as it's parameter.

CHAPTER

THREE

REFERENCE DOCUMENTATION

3.1 Route handling

pyroutes.**route**(*path*)
Decorator for declaring a method as a handler for a specific path, and paths above it if no better match exists. The decorated method should take an instance of pyroutes.http.request.Request as its first argument, and optionally more arguments auto-populated from the path.

All handler methods must return an instance of pyroutes.http.response.Response.

pyroutes.**reverse_url**(*handler*)
Return the path a handler is responible for.

Example:

```
@pyroutes.route('/foo/bar')
def foo_method(request):
    return "foo"

pyroutes.reverse_url('foo_method')
'/foo/bar'
```

3.2 Request and response objects

3.2.1 Overview

All handlers and middleware receive an instance of the Request class and must return an instance of the Response class.

This document explains the APIs for Response and Request.

3.2.2 Request objects

class pyroutes.http.request.**Request**

Attributes

Request.**GET**
A dictionary of the given GET variables.

`Request.`**`POST`**
A dictionary of the given `POST` variables. If a multiple key-value pairs has the same key name, the dictionary value for the key will be a list of the values.

`Request.`**`FILES`**
A dictionary of the given files. As with the `POST` variables, multiple keys with the same name will result in a list. The values are tuples consisting of the filename and a file-like object.

`Request.`**`COOKIES`**
An instance if `RequestCookieHandler` which provides the following API.

`get_cookie`(*key*)
Return the value of a signed cookie, or None if it doesn't exist. Raises `CookieHashInvalid` or `CookieHashMissing` if the user has altered the cookies in any way.

`get_unsigned_cookie`(*key*)
Return the value of a cookie without validating it's authenticity. Returns None if the cookie doesn't exist.

`Request.`**`ENV`**
The environment as passed from `WSGI`.

3.2.3 Response objects

`Response([content=None, headers=None, status_code='200 OK', default_content_header=True])`

The base response class. Constructor initializes the attributes with its given values. If `default_content_header` is true, and `Content-Type` is not passed in headers, the content type defined in `pyroutes.settings.DEFAULT_CONTENT_TYPE` will be added to the headers automatically.

The status code can be passed as `either` a full status code `or` an integer corresponding to a standard code.

Attributes

`Response.`**`content`**
A string or an iterable object that is passed to the browser.

`Response.`**`status_code`**
The HTTP status code sent to the client. Can be either a full string representation of the status code, or just the number id.

`Response.`**`headers`**
A list of tuples with key-value pairs of headers and their value.

`Response.`**`cookies`**
An instance of `ResponseCookieHandler` which provides the following API.

`add_cookie`(*key*, *value*[, *expires=None*])
Adds a signed cookie to the response. The `expires` parameter must be an instance of `datetime.datetime` and set the cookie expiration to its value. Defaults to infinite lifetime.

`add_unsigned_cookie`(*key*, *value*[, *expires=None*])
Same functionality as `add_cookie` only the cookie will not be signed, and is not tamper proof.

`del_cookie`(*key*)
Deletes a cookie from the browser.

3.2.4 Redirect objects

class pyroutes.http.response.**Redirect**(*location*[, *absolute_path=False*])

A redirect shortcut class for redirection responses. This class can make two types of redirects:

1. Absolute path redirects: When you want do redirect to outside your application.
2. Root App relative redirect: If you want your redirection relative to the root application path.
3. Site relative redirect: If you want your redirection relative to the base of the domain.
4. Current URL relative redirect: If you want your redirection relative to the current URL value.

Example of these uses, assuming your app is installed on `http://example.com/apps/myapp`:

1. `Redirect('http://pyroutes.com/')` for a redirect to pyroutes.com.
2. `Redirect('/some/path')` for a redirect to `http://example.com/apps/myapp/some/path`. Ignores current URL.
3. `Redirect('/other/path', absolute_path=True)` to redirect to `http://example.com/other/path`. Ignores current URL.
4. `Redirect('relative/path')` to redirect to a path relative to the current URL of the user's browser. E.g. if the user is visiting `http://example.com/apps/myapp/foo/` this redirect would go to `http://example.com/apps/myapp/foo/relative/path`

3.2.5 HttpException objects

class pyroutes.http.response.**HttpException**(*location*[, ***template_data*])

Base class for all exceptions that produce special error pages. If instances of objects that inherited from HttpException are raised, the ErrorHandlerMiddleware will render a page and return it with the correct HTTP code. The base template can be overriden using settings.CUSTOM_BASE_TEMPLATE and additional template data kan be passed to the exception. Raising a HttpException without passing a location as first parameter is allowed, location is then populated from the PATH_INFO variable.

class pyroutes.http.response.**Http403**

Template for this HttpException can be overridden using settings.TEMPLATE_403

class pyroutes.http.response.**Http404**

Template for this HttpException can be overridden using settings.TEMPLATE_404

class pyroutes.http.response.**Http500**

Template for this HttpException can be overridden using settings.TEMPLATE_500

3.3 Templates

TemplateRenderer([base_template=None, inclusion_param=None, template_dir=None])

Creates a TemplateRenderer for doing single-inheritance rendering with XML-Template.

Takes three parameters.

- `base_template`: The base template. Usually with the html headers, css, etc.
- `inclusion_param`: The <t:id=”..”/> id which will be replaced with the content of the child template.

- `template_dir`: Override the `TEMPLATE_DIR` setting from pyroutes.settings.

`pyroutes.templates.`**`render`**(*template*, *data*)
Renders a template and returns the result as a string. Takes two parameters, the child template to be rendered, and the data passed to the templates.

3.4 Utilities

This module holds handy utilities for development and debugging of pyroutes applications. The methods found here are _not_ meant for use in production environments, no guarantees are given for their stability or security.

`pyroutes.utils.`**`devserver`**(*application*, *port=8001*, *address='0.0.0.0'*, *auto_reload=True*)
Simple development server for rapid development. Use to create a simple web server. For testing purposes only. Has no builting handling of file serving, use fileserver in this class. Typical usage:

```
from pyroutes import route, application, utils
#<define routes>
if __name__ == '__main__':
    utils.devserver(application)
```

This starts a server listening on all interfaces, port 8001. It automatically reloads modified files.

`pyroutes.utils.`**`fileserver`**(*request*, **path_list*)
Simple file server for development servers. Not for use in production environments. Typical usage:

```
from pyroutes import route, utils
route('/media')(utils.fileserver)
```

That will add the fileserver to the route /media. If DEV_MEDIA_BASE is defined in settings, host files from this folder. Otherwise, use current working directory.

Note: DEV_MEDIA_BASE and route path is concatenated, i.e. if you use '/srv/media' for as the media base, and map the route to '/files', all files will be looked for in '/srv/media/files'

3.5 Settings

Pyroutes enables you to have per project settings by overriding `pyroutes.settings` in a file called `pyroutes_settings.py` in your `PYTHON_PATH`.

The following settings are defined and can be overridden:

`DEBUG`
Enables debugging output on the `500 Server Error`. Should not be true in production, as it might leak sensitive data.

Default: False

`DEFAULT_CONTENT_TYPE`
The default content type set by `Response`.

Default: *'text/html; charset=utf8'*

`SECRET_KEY`
Key for cryptographic functions in pyroutes. You *must* override this in your per project settings for the crypto to add any value.

TEMPLATE_DIR
The location of the filesystem `TemplateRenderer` looks for the templates. If set to None it will try to load from the current working directory.

Default: None

MIDDLEWARE
A list of the enabled middlewares to be run, from outer to inner. You most likely want to keep the default middleware classes if enabling more middleware. Also, note that for the ErrorHandlerMiddleware to handle errors it *must* be the last element of the list.

You can read more about middleware here *Middleware*.

Default: `('pyroutes.middleware.errors.NotFoundMiddleware', 'pyroutes.middleware.errors.ErrorHandlerMiddleware',)`

SITE_ROOT
This setting governs the behaviour of the Redirect class.

Any redirect that isn't absolute will be relative to this path. E.g. if an application is set up at http://example.com/pyroutes/demo/ then as default a Redirect('/foo') will go to /pyroutes/demo/foo/. If the SITE_ROOT variable is set to the empty string, the redirect goes to /foo/ and if SITE_ROOT is set to '/bar' the redirect goes to /bar/foo/.

Default: `Detected automatically from environment`

CUSTOM_BASE_TEMPLATE
Used for custom HttpException base template. See the default template for an example. Defining only page templates and not the base template is allowed.

TEMPLATE_403 = './templates/403.xml'
Used for the rendering pages when a Http403 is raised.

TEMPLATE_404 = './templates/404.xml'
Used for the rendering pages when a Http404 is raised.

TEMPLATE_500 = './templates/500.xml'
Used for the rendering pages when a Http500 is raised.

CHAPTER

FOUR

APPENDICES

4.1 Introduction to XML::Template

Note: This is shamelessly ripped from http://bzr.sesse.net/xml-template/doc/intro.txt and converted to RST. The examples are in Perl, but the syntax is close to identical.

XML::Template is a templating system; there are already many others, so if you do not like it, look into a different one (its design is inspired by at least Template Toolkit and Kid, probably also including elements from others). XML::Template is (like Kid or TAL) designed to guarantee that your output is well-formed XML, which is a good step on the road to give you valid XHTML.

You can get the latest version of XML::Template with bzr; get bzr from your favourite distribution and do:

```
$ bzr get http://bzr.sesse.net/xml-template/
```

to check out the code and this documentation.

There is a lot to be said about design philosophy, but let's first give a simple example to give you the feel of how it works. (The example is in Perl, but there are also functionally equivalent PHP, Python and Ruby versions; ports to other languages would be welcome.)

Template (simple.xml)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE
  html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://template.sesse.net/" xml:lang="en">
  <head>
    <title />
  </head>
  <body>
    <p t:id="hello">This will be replaced.</p>
  </body>
</html>
```

Code (simple.pl)

```
#! /usr/bin/perl
use XML::Template;

my $doc = XML::Template::process_file('../xml/simple.xml', {
    'title' => 'A very basic example',
```

```
    '#hello' => 'Hello world!'
});
print $doc->toString;
```

Result

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>A very basic example</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

This is about as simple as it gets, but we've already touched on most of the functionality we want or need. A few points are worth commenting on:

- We template by first _selecting_ certain elements (either by tag name, or by ID – the syntax is borrowed from CSS since you probably already know it), then _replace_ their contents. (Soon, we'll also _clone_ elements.)
- We get a DOM tree out, which we can either print out or do other things with (say, style further if XML::Template should not prove enough). (Actually, we start with a DOM tree as well, but process_file is a shortcut to read in an XML file and parse it into a DOM tree first, since that's usually what we want.)
- All traces of our templating system have been removed – there is a flag you can give to prohibit this "cleaning" in case you don't want that.

Note how little syntax we need to do simple things – XML::Template is designed to *keep simple things simple*, since you want to do simple things most of the time. (I don't believe in "lines of code" as the primary metric for API usability in general, though.)

We move on to another useful operation, cloning.

Template (clone.xml)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE
  html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://template.sesse.net/" xml:lang="en">
  <head>
    <title>Cloning test</title>
  </head>
  <body>
     <p>My favourite color is <t:color />; I like that very much.
       All my favourite things:</p>
    <ul t:id="things">
      <li />
    </ul>
  </body>
</html>
```

Code (clone.pl)

```
#! /usr/bin/perl
use XML::Template;

my $doc = XML::Template::process_file('../xml/clone.xml', {
```

```
    'color' => 'blue',
    '#things' => [
        { 'li' => 'Raindrops on roses' },
        { 'li' => 'Whiskers on kittens' },
        { 'li' => 'Bright copper kettles' },
        { 'li' => 'Warm, woolen mittens'}
    ]
});
print $doc->toString;
```

Result

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>Cloning test</title>
  </head>
  <body>
     <p>My favourite color is blue; I like that very much.
       All my favourite things:</p>
    <ul>
      <li>Raindrops on roses</li>

      <li>Whiskers on kittens</li>

      <li>Bright copper kettles</li>

      <li>Warm, woolen mittens</li>
    </ul>
  </body>
</html>
```

This isn't much harder than the example above; we've done a simple replacement of the contents of <t:color> to "blue" (and after that just removed the tag; any tag you use in the templating namespace will automatically get stripped away), and then cloned the contents of our "things" bullet list. Note that XML::Template automatically recurses after the cloning, since you probably don't want four identical elements. You can recurse as many times as you'd like, in case you'd need lists of lists or multiple tables and rows and columns – you don't even have to understand what's happening to get it to work.

Note that we did all of this without any logic in the template at all. This is completely intentional – it's a bit of an experiment, really, but hopefully it will all turn out well. There is no logic in the templating system at all; if-s are handled with replacements (or DOM deletions), for-s are handled with cloning and expressions are handled by the language you're using.

This means we have introduced all three operations we need (replacement, substitution/selection and repeating/cloning), and only need two more features before we're all done.

The first one is just a variation on replacement; instead of replacing with a string, you can replace with a DOM tree or document. This facilitates simple inclusion, since you probably want some header and footer to be the same across all your pages. (No example here, you can probably work it out by yourself; just send a DOM object instead of a string. There's an example in the source code distribution if you need it.)

The second one is also a variation on replacement; sometimes, you want to set attributes on elements instead of replacing their contents, and for that, we have a small hack:

Template (clone.xml), repeated for your convenience

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE
  html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://template.sesse.net/" xml:lang="en">
  <head>
    <title>Cloning test</title>
  </head>
  <body>
    <p>My favourite color is <t:color />; I like that very much.
      All my favourite things:</p>
    <ul t:id="things">
      <li />
    </ul>
  </body>
</html>
```

Code (attribute.pl)

```
#! /usr/bin/perl
use XML::Template;

my $doc = XML::Template::process_file('../xml/clone.xml', {
    'color' => 'red',
    '#things' => [
        { 'li' => 'Raindrops on roses',    'li/class' => 'odd' },
        { 'li' => 'Whiskers on kittens',   'li/class' => 'even' },
        { 'li' => 'Bright copper kettles', 'li/class' => 'odd' },
        { 'li' => 'Warm, woolen mittens',  'li/class' => 'even' }
    ]
});
print $doc->toString;
```

Result

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>Cloning test</title>
  </head>
  <body>
     <p>My favourite color is red; I like that very much.
       All my favourite things:</p>
    <ul>
      <li class="odd">Raindrops on roses</li>

      <li class="even">Whiskers on kittens</li>

      <li class="odd">Bright copper kettles</li>

      <li class="even">Warm, woolen mittens</li>
    </ul>
  </body>
</html>
```

Naturally, you can't put anything else than a simple string into an attribute, but it's not like this is a big limitation. (There's also a shortcut for doing stuff like odd/even automatically, but I'll leave that for yourself to find out; see the attribute2 example.)

That's it for the examples; now let's turn to the boring design philosophy.

The main thoughts behind XML::Template have been, in no particular order:

- Make the simple things simple. (A template should not be much more cumbersome to write than if you wrote the page statically.) More complex things can be harder if it makes the simple things simpler; that's OK.
- Make it easy for the user to do the right thing. (Guarantee well-formed XML, and make a design that makes it easy to separate back-end logic, viewing logic and HTML templating. Incidentially, I've only seen one library ever that does the same properly for database logic and other back-end logic, and that is the excellent libpqxx library.)
- Premature optimization is the root of all evil; most web systems are not performance limited by their output anyway.
- Don't try to be everything for everyone. (XML::Template can not output to plain text or PostScript, even though that would clearly be useful for some people in some cases.)
- Be language agnostic. (DOM is rather universal, and there's a useful implementation for most web-relevant languages out there.) Maintaining several implementations in several languages is suboptimal, but it's better than only supporting one language or having someting that needs to reimplement the entire DOM with wrappers for each language. (Thankfully, by relying on the DOM support in each language, the code so far is under 200 lines per implementation, so maintaining this hopefully shouldn't be much work.) As proof-of-concept, there are got Perl, PHP, Python and Ruby implementations that work and feel largely the same (and even a SAX-based Perl implementation, for larger trees that won't fit into memory) – other implementations are welcome. This is backed up by a test suite, which ensures that all the different implementations return structurally equivalent XML for a certain set of test cases. Porting to a new language is not difficult, and once you've got all the test cases to pass, your work is most likely done.

As a side note to the second point, I've spent some time wondering exactly _why_ you want to separate the back-end logic from your HTML, and why people don't seem to do it. After some thought, I've decided that what I really want is to get the HTML away from my code – not the other way round. (In other words, HTML uglifies code more then code uglifies HTML – someone using a WYSIWYG editor to generate their HTML might disagree, though.)

However, this also means that you want the _entire_ viewing logic away from your back-end logic if you can. When you process your data, you really don't want to care if you're on an odd or even row to get those styled differently in the HTML; that's for another part. XML::Template, incidentially, by moving the entire output logic to the end of your script, makes this easy for you; you *can* do the viewing logic "underway" if you really want to, but there's no incentive to, and the natural modus operandi is to split viewing and other logic into two distinct parts.

An open question is how to do internationalization on web pages; I haven't yet seen a good system for handling this. To be honest, this might be something handled in another layer (cf. "don't try to be everything to everyone" above), but I'd be interesting to hear others' thoughts on this, especially how you could achieve clean text/markup separation (stuff like gettext doesn't really work well with markup in general).

More to come here at some point, probably. Now, go out and just _use_ the thing – I hope it will make your life on the web simpler. :-)

- Steinar H. Gunderson <sgunderson@bigfoot.com>, http://www.sesse.net/

CHAPTER

FIVE

CONTRIBUTING

The source code is version controlled using git and resides on http://github.com/pyroutes/pyroutes. Feel free to clone it and fix stuff :-) Just remember to check if the test-suite still passes. I will not pull in features if they don't include documentation, unittest or simply breaks the existing tests. Bugfixes to the tests are always welcome too though.

For the tests to run, you will need some extra packages. Names are by debian/ubuntu package name.

- python-coverage
- python-nose

PYTHON MODULE INDEX

p