Pyxer 0.6.1

Yet another Python Framework

(C)opyright by Dirk Holtwick, Germany
dirk.holtwick@gmail.com
http://www.pyxer.net

Introduction

The **Pyxer** Server is a very simple Python Web Framework that aims to makes starting a new project as easy as it can be. It still works respecting the MVC concept but the files can be mixed in one directory. For a high end solution you should maybe better consider using Pylons, Django, TurboGears and similar.

This work is inspired by http://pythonpaste.org/webob/do-it-yourself.html.

Technical background

The Google App Engine (GAE) in version 1.1 offers a very restricted Python environment and the developer as to ship arround a lot of problems. Pyxer helps on this point by also providing solutions that work with the great WSGI Framework Paster by Ian Bicking. So you get the best from both sides: GAE and Paster. To achieve this some other great third party tools are used like WebOb and VirtualEnv (also by Ian Bicking) and for templating the html5lib is used.

Installation

Install Pyxer using easy_install:

```
$ easy_install pyxer
```

All required packages (webob, html5lib, beaker) should be installed automatically if needed.

If you want to use Google AppEngine install it separately too.

Quick tutorial

Create a new project

At first set up a new Pyxer project using the Pyxer command line tool like this:

```
$  pyxer init myexample
```

In the newly created directory "myexample" you will find a directory structure like this one (under Windows "bin" will be called "Scripts"):

```
bin/
public/
lib/
```

Place your files in the "public" directory.

Start the server

To start the server you may choose between the Paster-Engine:

```
$ xpaster serve
```

or the GAE-Engine:

```
$ xgae serve
```

Or use Pyxer command line tool again to use the default engine (which is WSGI from Python standard lib):

```
$ pyxer serve
```

But you may also use Pyxer without using the command line tools e.g. like this:

```
$ paster serve development.ini
```

"Hello World"

For a simple "Hello World" just put an "index.html" file into the "public" directory with the following content:

```
Hello World
```

This works just like a static server. To use a controller put a file "__init__.py" into that directory with the following content:

```
@controller
def index():
   return "Hello World"
```

Controllers

Controller, templates and static files are placed in the same directory (usually "public"). First **Pyxer** looks for a matching controller. A controller is defined in the "__init__.py" file and decorated by using `@controller` which is defined in `pyxer.base`.

```
from pyxer.base import *

@controller
def index():
   return "Hello World"
```

@expose

This controller adds the GET and POST parameters as arguments to the function call (like in CherryPy):

```
from pyxer.base import *

@expose
def index(name="unknown"):
   return "Your name is " + name
```

default()

If the matching controller can not be found the one called "default" will be called:

```
from pyxer.base import *

@controller
def default():
   return "This is path: " + request.path
```

Templates

This example can be called like `/` or `/index`. To use a Pyxer template with this file you may use the `render()` function or just return `None` (that is the same as not returning anything) and the matching template will be used, in this case `index.html`. The available object in the template are the same as used by Pylons: `c` = context, `g` = globals and `h` = helpers.

```
from pyxer.base import *

@controller
def index():
   c.title = "Hello World"
```

By default the Kid templating language is used and output is specified as "xhtml-strict". You may want to change that for certain documents e.g. to render a plain text:

```
from pyxer.base import *

@controller(output="plain")
def index():
   c.title = "Hello World"
```

Or use another template:

```
from pyxer.base import *

@controller(template="test.html", output="html")
def index():
   c.title = "Hello World"
```

Or use your own renderer:

```
from pyxer.base import *

def myrender():
   result = request.result
   return "The result is: " + repr(result)

@controller(render=myrender)
def index():
   return 9 + 9
```

JSON

To return data as JSON just return a dict or list object from your controller:

```
from pyxer.base import *

@controller
def index():
  return dict(success=True, msg="Everything ok")
```

Sessions

Session are realized using the Beaker package. You may use the variable `session` to set and get values. To store the session data use `session.save()`. Here is a simple example of a counter:

```
from pyxer.base import *

@controller
def index():
   c.ctr = session.get("ctr", 0)
   session["ctr"] = c.ctr + 1
   session.save()
```

??? XXX

   1. #Looks for a controller (foo.bar:bar)

      1. If the controller returns a dictionary this will be applied to template (step 2)

2. Looks for the template (foo/bar.html)

Deployment

To publish your project to GAE you may also use the Pyxer command line tool. First check if your "app.yaml" file contains the right informations like the project name and version infos. Then just do like this:

```
$ xgae upload
```

Be aware that Pyxer first needs to fix the paths to be relative instead of absolute to make them work on the GAE environment. If you choose not to use Pyxer for uploading you have to do this fix up explicitly **before** you upload your applikation, like this:

```
$ xgae fix
```

Routing

The routing in Pyxer is based on some conventions by default but may be extended in a very flexible and easy way. By default all public project data are expected in the folder `public`. If you just put static content there it will behave like you expect it from a normal webserver.

To add controllers to it, start with creating an `__init__.py` file. This makes the folder a Python package and Pyxer routing will first evaluate this before looking for static content.

Default behaviour

The easiest controller looks like this:

```
@controller
def index():
   return "Hello World"
```

This will be called with the following URLs:

- `http://<domain>/`
- `http://<domain>/index`
- `http://<domain>/index.html`
- `http://<domain>/index.htm`

If the controller has another name as `index`, these corresponding URLs will match:

- `http://<domain>/<controller>`
- `http://<domain>/<controller>.html`
- `http://<domain>/<controller>.htm`

There is one other special controller name `default`. If this one exists all non matching request will be passed to this controller.

If you have sub packages in your `public` folder like `foo` and `foo.bar`, these will be matched by the corresponding path and in this package the rules described before will apply:

- `http://<domain>/foo/bar/...`

Everything that does not match will be considered static content. Pyxer tries to match the path relatively to the last matched package.

If you are just beginning to use Pyxer and you do not like reading manual - like I do - you may skip the next sub sections and continue with the next main section.

Custom routes

If you need more sophisticated routing or want to include external packages that are not placed under the `public` folder you may add your own routing. This is as simple as adding this line to your global space of your module:

```
router = Router()
```

Important! The name of this object has to be `router` by convention!

To add your own ...

```
router.add("content-{name}", "content")
```

This matches all URL starting with `content-` while the rest will be saved in `req.urlvars` as `name`. For example the URL `/content-myentry` will result in a call of the controller `index` where `req.urlvars["name"]=="myentry"`.

For more complicated routes you may also use the `add_re` method, which offers more flexibility. Here is an example that matches the rest of the path after "content/" and passes the value to the controller via `req.urlvars["path"]`:

```
router.add_re("^content/(?P<path>.*?)$",
 controller="index", name="_content")
```

Relative URL

In your templates you should try to often use the `h.url()` helper. It calculates a URL relative to the matched routes base. For example if we take a look at the `add_re` routing example we can see that the `path` component is under the content component. Let's say we have a controller called `edit` we like to call from the page created by `index`, then we can not write it like this:

```
<a href="edit?path=$c.path">Edit</a>
```

That does not always work because this page could have been called via "index?path=xyz" or via "content/xyz". To make sure we are get the URL corresponding to our modules controller we could write it like this:

```
<a href="${h.url('edit?path=' + c.path)}>Edit</a>
```

Or even better using the feature of `h.url` that lets you append GET parameters as named arguments of the helper function:

```
<a href="${h.url('edit', path=c.path)}>Edit</a>
```

If you use `redirect` it will call the `url` helper too so that relative parts will be translated to absolute ones.

Templating

Pyxer offers yet another templating language that is very close to Genshi and Kid. As the former did not work with Google AppEngine at the moment of birth of **Pyxer** the new templating tools had been implemented.

Variables and expressions

The default templating works similar to most known other templating languages. Variables and expressions are realized like `$<varname>` (where `<varname>` may contain dots!) and `${<expression>}`:

```
Hello ${name.capitalize()}, you won $price.
$item.amount times $item.name.
```

Commands

These are also known form templating languages like Genshi and Kid. They are used like this:

```
<div py:if="name.startswith('tom ')">Welcome $name</div>
```

Or this:

```
<div py:for="name in sorted(c.listOfNames)">Welcome $name</div>
```

These are the available commands. They behave like the Genshi equivalents:

- py:if ~~... py:else ... py:elif~~
- py:for ~~... py:else~~
- py:def
- py:match
- py:layout~~ / py:extends~~
- ~~py:with~~
- py:content
- py:replace
- py:strip
- py:attrs

Comments

If HTML comments start with "!" they are ignored form the output:

```
<!--! Invisible --> <!-- Visible in browsers -->
```

Layout templates

The implementation of layout templates is quite easy. Place the `py:layout` command in the `<html>` tag and pass a Template object. For loading you may use the convenience function `load()`.

```
<html py:layout="load('layout.html')">
 ...
</html>
```

In the template file you may then access the original template stream with the global variable `top`. Use CSS selection or XPATH to access elements. Example:

```
<html>
 <title py:content="top.css('title')"></title>
 <body>
   <h1><a href="/">Home</a>
       / ${top.select('//title/text()')}</h1>
   <div class="content">
     ${top.css('body *')}
   </div>
 </body>
</html>
```

XPath

XPath is supported like it is in Genshi.

CSS Selectors

CSS Selectors ending with " *" return just the inner texts and elements of the matched pattern.

Databases

You are free to use any database model you like. For GAE you have not much choice but for other engines I recommend using Elixir. You should try to separate your controller stuff from your database stuff by creating a Python module called "model.py". For a GAE project this may look like this:

```
from google.appengine.ext import db
from google.appengine.api import users

class GuestBook(db.Model):
  name = db.StringProperty()
  date = db.DateTimeProperty(auto_now_add=True)
```

While using Elixir you may do like this:

```
xxx
```

XXX See the GuestBook example for a complete demo.

Advanced

Python virtual environment

To make deployment of GAE projects easy a virtual environment (VM) is created. If you start GAE via `xgae` or paster via `xpaster` these virtual environments will automatically be used. Pyxer determines the root of the VM by looking for the `app.yaml` file. If you have to enter the VM for installing packages or for other reasons you may to it like this:

```
$ pyxer vm
(vm)$ easy_install html5lib
(vm)$ exit
```

You may also use the usual functions as described in virtualenv by Ian Bicking http://pypi.python.org/pypi/virtualenv/.

```
$ Scripts\activate.bat
$ easy_install SomePackageName
$ deactivate
```

And for other Unix like system like this:

```
$ source bin/activate
$ easy_install SomePackageName
$ deactivate
```

Development of Pyxer under GAE

If you decide to develop Pyxer you may run into the following problem: each project comes with an own virtual machine (VM)and its own installation of Pyxer in it. So if you change the development version it will have no effect on your installation. Therefore a command "pyxer" is added that synchronizes the Pyxer installation in the VM with the development version:

```
$ pyxer pyxer
```

BTW: To install the development version using SetupTools do like this:

```
$ cd <Path_to_development_version_of_Pyxer>
$ python setup.py develop
```

You will have to repeat this each time the version of Pyxer changes, because otherwise the command line tools do not work.

Writing test cases

Since a Pyxer project is based in Paster writing test cases is quite the same. The most simple test looks like this. (We asume that the test file to be placed in the root of the project. For normal testing you have do add root to `sys.path` and modify the `loadapp` argument.):

```
from paste.deploy import loadapp
from paste.fixture import TestApp
import os.path
```

```
app = TestApp(loadapp('config:%s' % os.path.abspath('development.ini')))
res = app.get("/")
assert ('<body' in res)
```

For more informations look here http://pythonpaste.org/testing-applications.html.

Use within Eclipse

xxx

Use Google App Engine Launcher on Mac OS

xxx

Pyxer on Apache

If you have installed mod_python the deployment of your project is as simple as the following five lines. Just copy them to your sites configuration and adjust the absolute path to the "development.ini":

```
<Location "/">
 SetHandler python-program
 PythonHandler paste.modpython
 PythonOption paste.ini /<absolute_path_to_ini_file>/development.ini
</Location>
```

Configuration

Pyxer configuration is placed in the configuration file used by Paster or GAE respectively `development.ini` or `gae.ini`. If both are not available Pyxer looks into `pyxer.ini`. Example:

```
[pyxer]
session = beaker
```

Engines

XXX

Pyxer uses support different so called "engines" to publish a project. Most of them need own configurations and a well prepare environment to work fine. These are very specific to each of these engines and **Pyxer** tries to make the setup as easy as possible

Common options:

- `--host=HOST` (default: 127.0.0.1)
- `--port=PORT` (default: 8080)

WSGI

```
$ pyxer serve
```

Paster

Options:

- `--reload` XXX

With the virtual machine:

```
$ xpaster serve --reload
```

Without the virtual machine:

```
$ paster serve development.ini
```

Google Appengine

```
$ xgae serve
```

Appendix

Reserved names

- `index`: Name of the root controller
- `default`: Name of the collecting controller
- `router`: Name of the routing object
- `session`: Session
- `req`, `request`
- `resp`, `response`
- `cache`
- `c`
- `g`
- `h`
- `config`

Links

1. http://code.google.com/p/pyxer/ [Pyxer Project Homepage]