Pyxer 0.6.2

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 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 limitations. Pyxer helps on this point by providing solutions that also work together with the WSGI Framework Paste by Ian Bicking. This way you get the best from both sides: GAE and Paste. To achieve this, some other common third party tools are used like WebOb and VirtualEnv also by Ian Bicking. The templating s based on Genshi.

Installation

Install Pyxer using **`easy_install`** from SetupTools:

```
$easy_installpyxer
```

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

If you want to use Google App Engine you have to install it separately.

Quick tutorial

Create a new project

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

```
$pyxerinitmyexample
```

In the newly created directory **`myexample`** you will find a directory structure like the following (on 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:

```
$xpasterserve
```

Or the GAE-Engine:

```
$xgaeserve
```

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

```
$pyxerserve
```

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

```
$pasterservedevelopment.ini
```

"Hello World"

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

```
HelloWorld
```

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

```
@controller
defindex():
return"HelloWorld"
```

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`.

```
frompyxer.baseimport*

@controller
defindex():
return"HelloWorld"
```

@expose

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

```
frompyxer.baseimport*

@expose
defindex(name="unknown"):
return"Yournameis"+name
```

default()

If no matching controller can be found the one named `default` will be called:

```
frompyxer.baseimport*

@controller
defdefault():
return"Thisispath:"+request.path
```

Templates

This example can be called via `/` 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 using `return`) and the matching template will be used, in this case `index.html`. The available objects in the template are the same as used by Pylons: `c` = context, `g` = globals and `h` = helpers.

```
frompyxer.baseimport*

@controller
defindex():
c.title="HelloWorld"
```

By default a Genshi like 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:

```
frompyxer.baseimport*

@controller(output="plain")
defindex():
c.title="HelloWorld"
```

Or use another template:

```
frompyxer.baseimport*
```

```
@controller(template="test.html",output="html")
defindex():
c.title="HelloWorld"
```

Or use your own renderer:

```
frompyxer.baseimport*

defmyrender():
result=request.result
return"Theresultis:"+repr(result)

@controller(render=myrender)
defindex():
return9+9
```

JSON

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

```
frompyxer.baseimport*

@controller
defindex():
returndict(success=True,msg="Everythingok")
```

Sessions

Session are realized using the Beaker package. You can 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:

```
frompyxer.baseimport*

@controller
defindex():
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:

```
$xgaeupload
```

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 application like this:

```
$xgaefix
```

Routing

The routing in Pyxer is based on some conventions by default but can be extended in a very flexible and easy way. By default all public project data is expected to be found 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 by creating the **`__init__.py`** file. This makes the folder become a Python package and Pyxer routing will at first evaluate this one before looking for static content.

Default behaviour

The easiest controller looks like this:

```
@controller
defindex():
return"HelloWorld"
```

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 named **`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.

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 key called `name`. For example the URL `/content-myentry` will result in a call of the controller `content` 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:

```
<ahref="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 correct URL corresponding to our modules controller we could write it like this:

```
<ahref="${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:

```
<ahref="${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. Beause Genshi did not work with Google App Engine when **Pyxer** was started, the new templating tools have 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${name.capitalize()},youwon$price.
$item.amounttimes$item.name.
```

Commands

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

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

Or this:

```
<divpy:for=”nameinsorted(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 for output:

```
<!--!Invisible--><!--Visibleinbrowsers-->
```

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 can use the convenience function `load()`.

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

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

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

XPath

XPath is supported like it is in Genshi.

CSS Selectors

CSS Selectors ending with`*` (notice the space) 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 do not have 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:

```
fromgoogle.appengine.extimportdb
fromgoogle.appengine.apiimportusers

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

While using Elixir you can start 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 do it like this:

```
$pyxervm
(vm)$easy_installhtml5lib
(vm)$exit
```

You can also use the usual functions as described in virtualenv by Ian Bicking.

```
$Scripts\activate.bat
$easy_installSomePackageName
$deactivate
```

And for other Unix like system like this:

```
$sourcebin/activate
$easy_installSomePackageName
$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:

```
$pyxerpyxer
```

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

```
$cd<Path_to_development_version_of_Pyxer>
$pythonsetup.pydevelop
```

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 on Paster, writing test cases is quite the same. The most simple test looks like this. (We asume that the test file to will be placed in the root of the project. For normal testing you have do add the root directory, where **`app.yaml`** is placed, to **`sys.path`** and modify the **`loadapp`** argument.):

```
frompaste.deployimportloadapp
frompaste.fixtureimportTestApp
importos.path

app=TestApp(loadapp('config:%s'%os.path.abspath('development.ini')))
```

```
res=app.get("/")
assert('<body'inres)
```

For more informations about testing 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 writing the following five lines. Just copy them to your sites configuration and adjust the absolute path to the `development.ini`:

```
<Location"/">
SetHandlerpython-program
PythonHandlerpaste.modpython
PythonOptionpaste.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

```
$pyxerserve
```

Paster

Options:

- `--reload` XXX

With the virtual machine:

```
$xpasterserve--reload
```

Without the virtual machine:

```
$pasterservedevelopment.ini
```

Google App Engine

```
$xgaeserve
```

Utilities

Using reCaptcha

To make websites safe against abuse Pyxer integrates support for reCaptcha. You have to register your web sites URL for free there to get a private and public key. To use them see the following demo:

```
frompyxerimportrecaptcha

CAPTCHA_PUBLIC="XXX"
CAPTCHA_PRIVATE="XXX"

@expose
defmyform(**kw):
#ThiscreatestheHTMLcodeforthecaptchainputfield
c.captcha=recaptcha.html(CAPTCHA_PUBLIC)
#Dosomemoreformstuffhere
pass
#Testifthecaptchahasbeencorrect
ifrecaptcha.test(CAPTCHA_PRIVATE):
#Dosomethinglikestoringthedataintothedatabase
pass
#Aftersuccessjumptoanotherpage
returnredirect("success_info")
```

The corresponding template `myform.html` could look like this:

```
<formaction="">
<!--addmoreinputfieldshere-->
$c.captcha
<inputtype="submit"/>
</form>
```

Date, time and time zone

Handling of timezones is tricky in Python and on GAE. Therefore I will give a little introduction how this can be done quite easy. First you should install pytz like this:

```
$pyxervm
(vm)$easy_installpytz
```

If you want to show a datetime from Googles datastore you can now do like this in your templates:

```
Localtime${h.strftime(entry.added,'%H:%M','Europe/Berlin')}
```

Image upload

Simple upload GAE:

```
@expose
defupload(image=None):
```

```
ifimageisnotNone:
Images(image=db.Blob(image.file.read())).put()
```

Simple out:

```
@expose
defshow():
image=Images.all().get()#???
ifimage.image:
response.headers['Content-Type']='image/jpeg'
response.body_file.write(image.image)
return"NOIMAGE"
```

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]