Plastic Documentation

Release 0.1.0

Hong Minhee

February 06, 2013

CONTENTS

Plastic is a Python web framework built on top of Werkzeug.

CHAPTER

ONE

GETTING STARTED

1.1 Hello world

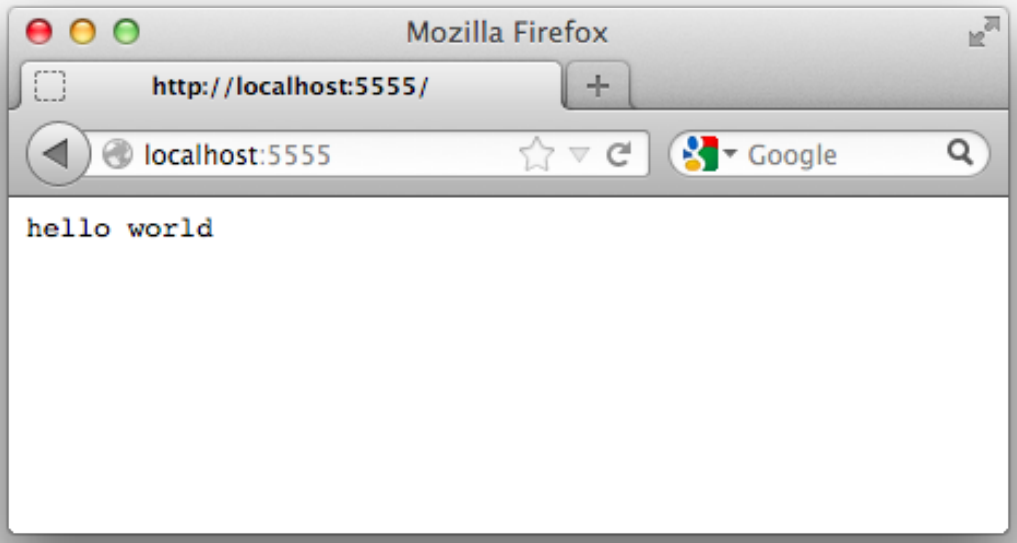
The following small code shows you how to print “hello world” using Plastic web framework:

```
from plastic.app import BaseApp

App = BaseApp.clone()

@App.route('/')
def hello(request):
    return 'Hello, world!'

if __name__ == '__main__':
    App().run()
```

Save the script (say it’s `hello.py`) and execute it:

```
$ python hello.py
 * Running on http://127.0.0.1:5555/
 * Restarting with reloader
```

Then open http://127.0.0.1:5555/ in your favorite web browser. The page says you “Hello, world!”:



1.2 How it was going on?

In the above example, we created our application class named `App`:

```
App = BaseApp.clone()
```

Yes, it's a class and also a subtype of `plastic.app.BaseApp`. It can do several things e.g. routing, service. Indeed it's a shorthand of the following "ordinary" inheritance code:

```
class App(BaseApp):
    pass
```

And then we defined a web page named `hello()`:

```
@App.route('/')
def hello(request):
    return 'Hello, world!'
```

We call these functions *view functions* (as like in Django or Flask). View functions can be routed using `BaseApp.route()` decorator, which is a convenient form of `BaseApp.add_rule()` method:

```
from werkzerug.routing import Rule

def hello(request):
    return 'Hello, world'
App.add_rule(Rule('/'), hello)
```

A view function has to take an argument at least, for getting a request. In a request object there are many information about a request e.g. HTTP method, HTTP headers, WSGI environment values.

It has to return a response. If it's simply a string, it will be a content of a response that *Content-Type* is *text/plain*. It also can return an instance of `plastic.message.Response`.

Lastly, the following line makes an app to be served:

```
if __name__ == '__main__':
    App().run()
```

The first `if` statement is a common idiom of Python to determine whether the file is directly executed by **python** as a script or imported by other module or script as a module. If it's imported by others as `hello` module, the code in the `if` statement will be ignored. If it's exeucted by **python** as a script (e.g. `python hello.py`), the code below will be also executed.

The `run()` method serves the app, so the above two lines mean: "run the web server if the file isn't imported by others but executed as a script." It's not a class method but an instance method, so it should be instantiated first: `App()`.

CHAPTER

TWO

DESIGN DECISIONS

2.1 Compared to “giants”

There have been several “giants” in web framework world. While Plastic inherited from those frameworks, there are key differences between Plastic and those frameworks.

2.1.1 Compared to Django

Explicit application instance Plastic applications have to be a subclass of `BaseApp` and can these make multiple instances of them. This means each application instance’s contexts and states are totally isolated from each other even in the same process. It makes applications easy to test.

No monolithic configuration Because of *explicit application instance*, each application instance has its own different configuration. So applications can be isolated from the environment. For example, you can make each configuration file for development, testing and production environments, and load the suitable configuration.

Persistence agnostic The one of main features Django provides is ORM. However, persistent objects can vary in production. You may want to use SQLAlchemy which is more powerful than Django ORM. There are some enthusiasts of ZODB as well. It’s the age of NoSQL. You may want to use Redis through Sider or MongoDB using MongoAlchemy. Moreover, some applications don’t have persist model objects *at all*. Some applications are more about *computations* than *data*.

Plastic is persistence agnostic and you can freely choose what you want to use for persistence.

Routing by decorators Django’s URL router requires the table to be assembled in `urls.py`:

```
from django.conf.urls.defaults import patterns, url

urlpatterns = patterns('',
    url(r'^$', 'myapp.home'),
    url(r'^people$', 'myapp.people'),
    url(r'^people/(?P<person_id>\d+)$', 'myapp.person')
)
```

Plastic uses decorators to route URL patterns to view functions:

```
@App.route('/')
def home(request):
    return '...'

@App.route('/people')
def people(request):
    return '...'
```

```
@App.route('/people/<int:person_id>')
def person(request, person_id):
    return '...'
```

You do not need switch between two or more files (`urls.py` and `views.py`) to map URL patterns to view functions.

Moreover, Plastic uses Werkzeug's routing system, so you have not to use regular expressions that are hard to read and not so suitable for URL patterns to route and can use Werkzeug's small URL pattern language that are easy to extend instead. (Compare `r'^people/(?P<person_id>\d+)$'` to `'/people/<int:person_id>'` in the above examples. :-)

2.1.2 Compared to Flask

Factory by default Flask suggests application factory pattern for larger applications, however it's not by default.

```
app = Flask(__name__)

@app.route('/')
def home():
    return 'hello'

if __name__ == '__main__':
    app.config['FOO'] = 'bar'
    my_superfast_wsgi_server(app)
```

The basic usage of Plastic `BaseApp` is subclassing. Routings and registering some hooks are done before application factory gets instantiated, and when it's instantiated configuration are applied to each instance.

```
App = BaseApp.clone()

@App.route('/')
def home(request):
    return 'hello'

if __name__ == '__main__':
    app = App(config={'FOO': 'bar'})
    my_superfast_wsgi_server(app)
```

You'll get factories in Plastic by default.

No context locals An interesting feature of Flask is the context locals. According to its design document, Flask chose context locals instead of explicit argument passing to make it quick and easy to write a traditional web application.

Technically it's similar to PHP's superglobals except PHP's contexts are completely isolated from each other while Flask's context isolation sometimes gets broken.

If Python had dynamic scoping like Common Lisp or Perl, context locals don't get so broken. Context locals are user-land implementation of dynamic scoping in programming languages that have no dynamic scoping. It's somewhat like magic, and magic isn't suitable for Python. :-(

Like Django and unlike Flask, Plastic takes the request argument as the first parameter for every view function. Everything is explicit. No magic.

If you look for `flask.g`, use `Request.context` attribute which is kept during each request context. `flask.current_app` becomes `Request.app`.

CHAPTER

THREE

PLASTIC — PLASTIC

3.1 plastic.app — Application factories

plastic.app.**DEFAULT_CONFIG** = ImmutableDict({'session_store': 'werkzeug.contrib.sessions:FilesystemSessionStore(filer
(collections.Mapping) The default configuration mapping.

class plastic.app.**BaseApp**(*config={}*)
The application base class. In order to make an application you have to inherit this class. As you know Python's subclassing can be done like:

```
from plastic.app import BaseApp

class MyApp(BaseApp):
    pass
```

However in most cases you don't have to fill its body but simply clone():

```
from plastic.app import BaseApp

MyApp = BaseApp.clone()
```

Use route() class method for routing.

classmethod add_rule(*rule*, *function*)
Adds a new rule and maps it to the function. It is a lower-level primitive method. In most cases you don't need it and just use route() decorator or other higher-level methods.

Parameters

- **rule** (werkzeug.routing.Rule) – the rule to add
- **function** (collections.Callable) – the function to map to the rule

classmethod add_serializer(*mimetype*, *function*)
Registers a function which serializes a value into a string. The funtion has to take two arguments and return its serialized result.

add_serializer.**function**(*request*, *value*)
Parameters
- **request** (Request) – the current request object
- **value** – a value to serialize into a string

Returns a serialized result
Return type basestring

Parameters

- **mimetype** (basestring) – a mimetype to assiciate the function with e.g. 'application/json'
- **function** (collections.Callable) – serializer function. see also function() for its signature

classmethod add_template_engine(*suffix*, *function*)

Registers a templating function to the given suffix. The function has to take three arguments and return its rendering result:

add_template_engine.**function**(*request*, *path*, *values*)

Parameters

- **request** (Request) – the current request object
- **path** (basestring) – the path of the template file. it can be a key of template_directory mapping object
- **values** (collections.Mapping) – the values to be passed to the template

Returns the rendering result

Return type basestring

Parameters

- **suffix** (basestring) – the filename suffix (without period character) to register the template engine e.g. 'mako'
- **function** (collections.Callable) – templating function. see also function() for its signature

classmethod associate_mimetypes(*mimetypes={}*, ***suffixes*)

Associates mimetype to suffix. Associations made by this are used by render() function. For example:

```
from plastic.rendering import render

App.associate_mimetypes({
    'text/plain': 'rst',
    'text/x-rst': 'rst',
    'text/html': 'html'
})

@App.route('/')
def home(request):
    return render(request, None, 'main/home')
```

with the above setting when you make a request like:

```
GET / HTTP/1.0
Accept: text/html
```

will find a template file main/home.html.* (* is suffix for template engine).

Parameters

- **mimetypes** (collections.Mapping) – a mapping object of mimetypes to suffixes e.g. {'text/html': 'txt', 'text/xml': 'xml'}
- ****suffixes** – keyword arguments are interpreted as suffixes to mimetypes. for example, passing txt='text/html' is equivalent to passing {'text/html': 'txt'}

classmethod clone(*__module__=None*, *__name__=None*, ***values*)

Subclasses the application class. It is a just shorthand of normal subclassing. If your application becomes bigger stop using this and normaly subclass the application class using class keyword instead.

config = None
(Config) Each application instance's configuration.

endpoints = {}
(collections.Mapping) The dict of endpoints to view functions.

mimetype_mapping = ImmutableDict({})
(ImmutableDict) The immutable dictionary of mimetype to registered renderers.

render_template (*request*, *path*, *values={}*, ***keywords*)
Renders the response using registered template_engines.

You have to pass the path *without specific suffix*. The last suffix will be used for determining what template engine to use. For example, if there is user_profile.html.mako file in the template_path and 'mako' is associated to Mako template engine in template_engines mapping, the following code will find user_profile.html.mako (not user_profile.html) and render it using Mako:

```
app.render_template(request, 'user_profile.html')
```

In other words, you have to append a suffix to determine what template engine to use into filenames of templates.

Parameters

- **request** (Request) – a request which make it to render
- **path** (basestring) – a path to template files without specific suffix
- **values** (collections.Mapping) – a dictionary of values to pass to template
- ****keywords** – the same to values except these are passed by keywords

Returns a rendered result

Return type basestring

Raises plastic.exceptions.RenderError when there are no matched template files

classmethod route (**rule_args*, ***rule_kwargs*)
The function decorator which maps the path to the decorated view function.

It takes the same arguments to the constructor of werkzeug.routing.Rule class. In most case simply give a path string to route:

```
@App.route('/people/<name>')
def person(request, name):
    return 'Hi, ' + name
```

rules = []
(collections.Sequence) The list of routing rules.

run (*host='127.0.0.1'*, *port=5555*, *debug=True*, ***options*)
Starts serving the application.

Parameters

- **host** (basestring) – the hostname to listen. default is '127.0.0.1' (localhost)
- **port** (int) – the port number to listen. default is 5555
- **debug** (bool) – use debugger and reloader. default is True
- ****options** – other options to be passed to werkzeug.routing.run_simple() function

classmethod serializer(*mimetypes*)
The function decorator which associate the given serializer `function` with `mimetypes`.

```
import json
import plistlib

@App.serializer('application/json')
def serialize_json(request, value):
    return json.dumps(value)

@App.serializer(['application/x-plist', 'text/xml'])
def serialize_plist(request, value):
    return plistlib.writePlistToString(value)
```

Parameters **mimetypes** (`basestring`, `collections.Iterable`) – one or more mimetypes to assiciate the `function` with. it can be either a single string or multiple strings in an iterable e.g. `'application/json'`, `['application/xml', 'text/xml']`

session_cookie
(`collections.Mapping`) The mapping of cookie settings sessions will use. It has the following configuable keys:

- `'key'` (`basestring`)
- `'max_age'` (`numbers.Integral`)
- `'domain'` (`basestring`)
- `'path'` (`basestring`)
- `'secure'` (`bool`)
- `'httponly'` (`bool`)

See Also:

Method **werkzeug.wrappers.BaseResponse.set_cookie()** Sets a cookie.

session_store
(`werkzeug.contrib.sessions.SessionStore`) The session store instance an application uses. It's a proxy to `'session_store'` value of `config`.

template_directory
(`ResourceDirectory`) The mapping object of the template directory.

classmethod template_engine(*suffix*)
The function decorator which makes the given `function` the template engine of the `suffix`.

```
# The following example has several issues.  Do not C&P
# it into your program to use Mako.
from mako.template import Template

@App.template_engine(suffix='mako')
def render_mako(request, path, values):
    with request.app.template_directory[path] as f:
        template = Template(f.read())
    return template.render(**values)
```

Parameters **suffix** (`basestring`) – the filename suffix (without period character) to register the template engine e.g. `'mako'`

template_engines = ImmutableDict({})
(ImmutableDict) The immutable dictionary of suffix to registered templating functions.

template_path = 'templates/'
(basestring) The path to the directory which contains template files. It has to be relative to the application module. It can be overridden. Default is templates/.

wsgi_app(*environ*, *start_response*)
The actual WSGI function. Replace it when the application should be wrapped by middlewares.

```
app.wsgi_app = Middleware(app.wsgi_app)
```

3.2 plastic.message — Request/response messages

class plastic.message.**Request**(*environ*, *app*, *populate_request=True*, *shallow=False*)
Bases: werkzeug.wrappers.Request

The richier subclass of werkzeug.wrappers.Request.

Parameters

- **environ** (collections.Mapping) – the wsgi environ to initialize with
- **app** (BaseApp) – the request app instance
- **endpoint** (basestring) – the requested endpoint

app = None
(BaseApp) The requested application instance.

bound_routing_map = None
(werkzeug.routing.MapAdapter) The bound url adapter.

build_url(*endpoint*, *_method=None*, *_external=False*, ***values*)
Builds an url for the given endpoint and other editional options.

It generates a relative path by default. Use _external=True when an absolute url is needed.

Parameters

- **endpoint** (basestring) – the endpoint of the url to build
- **_method** (basestring) – an optional HTTP method to disambiguate the rule
- **_external** (bool) – build the absolute url instead of relative url
- ****values** – the parameter values for the endpoint

context = None
(Context) The context storage, which is a kind of dict. Keys can be accessed as like attributes.

endpoint = None
(basestring) The requested endpoint.

endpoint_values = None
(collections.Mapping) The pararmeter values of routed endpoint.

session
(collections.MutableMapping) The session storage. If this value has changed in view functions the state will be kept in future requests of the same session as well.

class plastic.message.**Response**(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)

Bases: werkzeug.wrappers.Response

The richier subclass of werkzeug.wrappers.Response.

3.3 plastic.context — Contexts

class plastic.context.**Context**

Bases: dict, _abcoll.MutableMapping

The simple context storage which is a subtype of dict. It works in similar way except keys can be accessed via its attributes. For example, the following lines are:

```
assert isinstance(ctxt, plastic.context.Context)
ctxt.user_id = 123
user_id = ctxt.user_id
del ctxt.user_id
```

equivalent to:

```
ctxt['user_id'] = 123
user_id = ctxt['user_id']
del ctxt['user_id']
```

Note: You can't access reserved attribute names like dict.get(), dict.setdefault(). To use these names you have to use index operator:

```
ctxt['setdefault']
```

3.4 plastic.rendering — Content rendering

This module provides generic content rendering to make response using requested *Accept* mimetypes. In HTTP terminology, it's called as **content negotiation**.

Plastic provides two types of rendering methods:

Template engines It's for making responses that contains HTML. A typical case of use this is showing web pages to clients using their web browser.

Serializers It's for making responses that have structural data like JSON. A typical case of use this is providing RESTful API to clients.

plastic.rendering.**render**(*request*, *value*, *path*, *values={}*, ***keywords*)

Renders the suitable response using content negotiation. It's aware of given request's *Accept* header.

If there's no matched mimetype to request's *Accept* list, it raises NotAcceptable error.

If chosen mimetype is associated to serializer (see BaseApp.add_serializer()) it passes value to the serializer and rest of arguments are just ignored.

If chosen mimetype is associated to suffix (see BaseApp.associate_mimetypes() method) it appends the suffix to the given template path and then calls render_template() function with the same arguments except value.

Parameters

- **request** (Request) – a request which make it to render
- **value** – a value to serialize
- **path** (basestring) – a path to template files without suffix to determine content type and template engine
- **values** (collections.Mapping) – a dictionary of values to pass to template
- ****keywords** – the same to values except these are passed by keywords

Returns a rendered response

Return type Response

Raises

- **plastic.exceptions.RenderError** – when there are no matched template files
- **werkzeug.exceptions.NotAcceptable** – when there's no matched mimetype to request's *Accept* list

plastic.rendering.**render_template**(*request*, *path*, *values={}*, ***keywords*)

Helper function that renders a template of the given path with values (and keywords). It's a shortcut of BaseApp.render_template() method.

```
@App.route('/<int:user_id>')
def user_profile(request, user_id):
    global users
    user = users[user_id]
    return render_template(request, 'user_profile.html',
                           user_id=user_id, user=user)
```

You have to pass the path *without specific suffix*. The last suffix will be used for determining what template engine to use. For example, if there is user_profile.html.mako file in the template_path and 'mako' is associated to Mako template engine in template_engines mapping, the following code will find user_profile.html.mako (not user_profile.html) and render it using Mako:

```
render_template(request, 'user_profile.html')
```

In other words, you have to append a suffix to determine what template engine to use into filenames of templates.

Parameters

- **request** (Request) – a request which make it to render
- **path** (basestring) – a path to template files without suffix to determine template engine
- **values** (collections.Mapping) – a dictionary of values to pass to template
- ****keywords** – the same to values except these are passed by keywords

Returns a rendered result

Return type basestring

Raises **plastic.exceptions.RenderError** when there are no matched template files

See Also:

Method **BaseApp.render_template()** Renders the response using registered template_engines.

3.5 plastic.config — Configuration mapping

class plastic.config.**Config**
Bases: dict

Mapping object (subtype of dict) to store configurations.

update_from_file(*filename*, *overwrite=False*)
Updates configuration from Python file. For example, if there's dev.cfg:

```
debug = False
database_uri = 'sqlite://'
```

so you can load it using update_from_file() method:

```
config.update_from_file('dev.cfg')
```

Like update_from_object() method, it also ignores variables that start with underscore.

Parameters

- **filename** (basestring) – the path of Python file to load
- **overwrite** (bool) – keys that already exist are ignored by default. if True has given to this parameter, these are not overwritten

update_from_object(*object_*, *overwrite=False*)
Updates configuration from arbitrary object_.

```
@config.update_from_object
class default_config:
    debug = False
    database_uri = 'sqlite://'
```

It ignores attributes that start with underscore and keys that already exist until overwrite is True.

Parameters

- **object** – arbitrary object to update from, or import path of that if it's a string e.g. 'myapp.configs:prod
- **overwrite** (bool) – keys that already exist are ignored by default. if True has given to this parameter, these are not overwritten

update_unless_exists(*mapping=()*, ***keywords*)
Almost equivalent to update() except it ignores keys already exist.

```
>>> config = Config(a=1, b=2)
>>> config.update({'b': 1, 'c': 2})
>>> config
plastic.config.Config(a=1, b=2, c=3)
```

plastic.config.**config_property**
Maps application properties to configuration values.

plastic.config.**get_typename**(*cls*)
Finds the typename string of the given cls.

Parameters **cls** (type) – the class object to find its typename

Returns the typename

Return type basestring

plastic.config.**import_instance**(*expression*, *type_*)
This function provides a minimal language to import a class from a package/module and make an instance of it. For example, the following code:

```
val = import_instance('abc.defg:ClassName(3.14, hello, world=False)')
```

is equivalent to the following normal Python code:

```
from abc.defg import ClassName

val = ClassName(3.14, 'hello', world=False)
```

As you can see its syntax is slightly different from normal Python. You can pass arguments to class' constructor using its own syntax. You can pass limited types of values:

Booleans You can pass `True` and `False`.

Numbers It can take integers and floating numbers e.g. `123`, `3.14`.

Strings You can `'single quote'` and `"double quote"` both for string literals, and `r'raw string literals'` are also available. There are `u'Unicode string literals'` as well.

Moreover, if there re unquoted barewords these are also interpreted as strings.

None You can pass `None`.

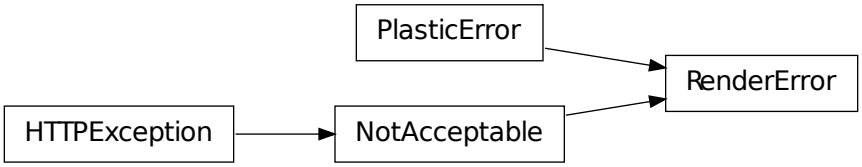
3.6 plastic.resourcedir — Resources directory

class plastic.resourcedir.**Resource**(*file_*)
Readable file object provided by `ResourceDirectory` mapping objects. You can treat this simply as file object. It also is a context manager, so you can use it using `with`.

Parameters **file** – file to wrap

class plastic.resourcedir.**ResourceDirectory**(*package*, *directory=''*)
Mapping interface of package resources.

3.7 plastic.exceptions — Exception types



exception plastic.exceptions.**PlasticError**
Bases: `exceptions.Exception`

All Plastic-related exceptions are derived from this class.

exception plastic.exceptions.**RenderError**(*message=None*, *description=None*)
Bases: plastic.exceptions.PlasticError, werkzeug.exceptions.NotAcceptable

All rendering-related errors are derived from this class. It also extends werkzeug.exceptions.NotAcceptable.

Parameters

- **message** – an optional error message for debugging purpose
- **description** – a message that will be showed to user agent

3.8 plastic.warnings — Warning categories

exception plastic.warnings.**AppWarning**
Bases: plastic.warnings.PlasticWarning

Warnings related to plastic.app module.

exception plastic.warnings.**PlasticWarning**
Bases: exceptions.Warning

The top-level warning category for Plastic-specific warnings.

3.9 plastic.version — Version data

plastic.version.**VERSION = '0.1.0'**
(str) The version string e.g. '1.2.3'.

plastic.version.**VERSION_INFO = (0, 1, 0)**
(tuple) The triple of version numbers e.g. (1, 2, 3).

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p