potpy Documentation

Release 0.0.1

David Zuwenden

January 20, 2013

CONTENTS

PotPy lets you build applications in a very flexible way. You design objects based on your domain requirements, then compose them together using PotPy's flexible routing system. PotPy enables you to:

- **Write simple, decoupled objects** that work together in various ways.
- **Easily test your objects**, by not imposing a rigid object construction paradigm.
- **Write WSGI components using TDD**, without having to deal with WSGI conventions except at the edges of the system.

PotPy was inspired by the Raptor project from the Ruby world.

CHAPTER

ONE

INSTALLATION

```
$ pip install potpy
```

For details, see the README file.

- GitHub project page
- PotPy on PyPI

CHAPTER

TWO

HELLO WORLD EXAMPLE

```
from potpy.wsgi import PathRouter, MethodRouter, App

# Our domain objects
# ------------------

class Greeter(object):
    """A WSGI app that displays a greeting."""
    def __init__(self, greeting):
        self.greeting = greeting

    def __call__(self, environ, start_response):
        start_response('200 OK', [
            ('Content-type', 'text/plain'),
            ('Content-length', str(len(self.greeting))),
        ])
        return [self.greeting]


def get_greeting(name):
    """Generate a greeting for the given name."""
    return 'Hello, %s' % (name,)



# PotPy plumbing
# --------------

hello = MethodRouter(
    (('GET', 'HEAD'), [                 # when the request is a GET or HEAD

        (get_greeting, 'greeting'),     # generate a greeting and save it
                                        # to the context under the
                                        # 'greeting' key

        Greeter                         # then show the greeting
    ]),
)


urls = PathRouter(
    ('hello', '/hello/{name}', hello),  # expose the greeter at /hello/{name}
)

```

```
44
45 if __name__ == '__main__':
46     from wsgiref.simple_server import make_server
47     make_server('', 8000, App(urls)).serve_forever()
```

CHAPTER

THREE

MORE EXAMPLES

- Todo List example

CHAPTER

FOUR

OVERVIEW

PotPy is a generic routing apparatus. It allows you to develop applications by composing your domain objects together into one or more "pipelines" for data to flow along. Composition is accomplished with the idea of a `Context` – return values from handler functions along these routes can be added to the context, enabling later handler functions to access them.

The magic happens when a context is *injected* into a callable. PotPy inspects the callable's signature, and pulls values out of the context by argument name. For example:

```
>>> from potpy.context import Context
>>> def my_callable(foo, bar, baz='default'):
...     # do something cool
...     return 42
...
>>> ctx = Context(foo='some value', bar='another value')
>>> ctx.inject(my_callable)
42
```

Building on the context, PotPy provides routes and routers. A `Route` is a list of callables that you define which get called in order. These callables are actually called by injecting a context, as above. The same context is used to call subsequent callables in the route, and these callables can either interact with the context directly, or their return value may be added to the context by the Route object. This allows later callables in a route to access information produced by earlier ones, which facilitates a very expressive style of application design.

A `Router` is an object that (typically) selects between various Routes given some condition. The Router class itself is an abstract base class, although the `potpy.wsgi` module provides two concrete subclasses that route based on specific WSGI *environ* variables (**PEP 333**). By subclassing the Router class and providing a `match()` method that selects based on something specific to your problem domain, you can build powerful control flows between the objects in your system with minimal effort.

The `potpy.template`, `potpy.wsgi`, and `potpy.configparser` modules turn PotPy into a flexible HTTP request routing system. `Template` objects allow string matching with parameter extraction, and the reverse – filling parameters into a string from a mapping. The `potpy.wsgi.PathRouter` class utilizes these templates to make URL-based routing convenient and easy. The `configparser` module enables you to specify a web application's URL layout in a simple declarative syntax, while being flexible enough to let you specify which HTTP methods (eg. *GET*, *POST*, etc.) your domain objects should handle, and exception handlers.

CHAPTER

FIVE

MODULE LISTING

5.1 potpy.context – Context module

5.1.1 Module Contents

class potpy.context.**Context**

A dict class that can call callables with arguments from itself.

Best explained with an example:

```
>>> def answer(question, foo):
...     return 'The answer to the %s question is: %d' % (question, foo)
...
>>> ctx = Context(foo=42, question='ultimate')
>>> ctx.inject(answer)
'The answer to the ultimate question is: 42'
```

Callable items are called before being passed to the callable:

```
>>> ctx = Context(foo=lambda bar: bar.upper(), bar='qux')
>>> ctx.inject(lambda foo: foo)
'QUX'
```

Note: Callable items are called during `__getitem__()`:

```
>>> Context(foo=lambda: 42)['foo']
42
```

Contexts have `'context'` as an implicit a member, so callables can refer to the context itself:

```
>>> ctx = Context(foo='foo')
>>> ctx.inject(lambda context: dict(context))
{'foo': 'foo'}
```

When injecting a call, you may override context items (or provide missing items) with keyword arguments:

```
>>> ctx.inject(lambda foo, bar: (foo, bar), bar='bar')
('foo', 'bar')
```

Note: `*args`- and `**kwargs`-style arguments cannot be injected at this time.

Note: Due to limitations of the `inspect` module, builtin and extension functions cannot be injected. You may work around this by wrapping the function in Python:

```
>>> ctx = Context(n='42')
>>> ctx.inject(lambda n: int(n))
42
```

---

inject (*func*, ***kwargs*)
Inject arguments from context into a callable.

Parameters

- **func** – The callable to inject arguments into.
- ****kwargs** – Specify values to override context items.

5.2 potpy.router – Router module

5.2.1 Module Contents

class potpy.router.**Route** (**handlers*)
A list of handlers which can be called with a `Context`.

Initializer can also be called with a single (non-tuple) iterable of handlers. Each handler item is either a callable or a tuple: `(handler, name, exception_handlers)` – see `add()` for details of this tuple.

__call__ (*context*)
Call the handlers in the route, in order, with the given context.

add (*handler*, *name=None*, *exception_handlers=()*)
Add a handler to the route.

Parameters

- **handler** – The "handler" callable to add.
- **name** – Optional. When specified, the return value of this handler will be added to the context under `name`.
- **exception_handlers** – Optional. A list of `(types, handler)` tuples, where `types` is an exception type (or tuple of types) to handle, and `handler` is a callable. See below for example.

Exception Handlers

When an exception occurs in a handler, `exc_info` will be temporarily added to the context and the list of exception handlers will be checked for an appropriate handler. If no handler can be found, the exception will be re-raised to the caller of the route.

If an appropriate exception handler is found, it will be called (the context will be injected, so handlers may take an `exc_info` argument), and its return value will be used in place of the original handler's return value.

Examples:

```
>>> from potpy.context import Context
```

```
>>> route = Route()
>>> route.add(lambda: {}['foo'], exception_handlers=[
...     (KeyError, lambda: 'bar')
... ])
>>> route(Context())
'bar'

>>> def read_line_from_file():
...     raise IOError()     # simulate a failed read
...
>>> def retry_read():
...     return 'success!'   # simulate retrying the read
...
>>> def process_line(line):
...     return line.upper()
...
>>> route = Route()
>>> route.add(read_line_from_file, 'line', [
...     # return value will be added to context as 'line'
...     ((OSError, IOError), retry_read)
... ])
>>> route.add(process_line)
>>> route(Context())
'SUCCESS!'

>>> route = Route()
>>> route.add(lambda: {}['foo'], exception_handlers=[
...     (IndexError, lambda: 'bar') # does not handle KeyError
... ])
>>> route(Context())    # so exception will be re-raised here
Traceback (most recent call last):
    ...
KeyError: 'foo'
```

previous

Refer to result of previous handler in route.

Example:

```
>>> from potpy.context import Context

>>> class MyClass:
...    def foo(self):
...        return 42
...
>>> route = Route(
...     MyClass,            # instantiate MyClass
...     Route.previous.foo  # refer to foo attribute of instance
... )
>>> route(Context())
42
```

context

Refer to a context item in route.

Example:

```
>>> from potpy.context import Context

>>> class MyClass:
```

```
...     def foo(self):
...         return 42
...
>>> route = Route(
...     Route.context.inst.foo  # refer to ctx['inst'].foo
... )
>>> route(Context(inst=MyClass()))
42
```

exception `Stop`(*value=<class 'potpy.router.NoValue'>*)

Raise this exception to jump out of a route early.

If an argument is provided, it will be used as the route return value, otherwise the return value of the previous handler will be returned.

Example::

```
>>> from potpy.context import Context


>>> def stopper():
...     raise Route.Stop('stops here')
...
>>> def foobar():
...     return 'never gets run'
...
>>> route = Route(stopper, foobar)
>>> route(Context())
'stops here'
```

class `potpy.router.Router`(**routes*)

Routes objects to handlers via a `match()` method.

When called with a `Context` and an object, that object will be checked against each registered handler for a match. When a matching handler is found, the context is updated with the result of the `match()` method, and the handler is called with the context.

Handlers are wrapped in `Route` objects, causing the context to be injected into the call. You may also add Route objects directly.

The `match()` method of this class is unimplemented. You must subclass it and provide an appropriate match method to define a Router. See `potpy.wsgi.PathRouter` and `potpy.wsgi.MethodRouter` for example subclasses.

`__call__`(*context*, *obj*)

Route the given object to a matching handler.

Parameters

- **context** – The `Context` object used when calling the matching handler.
- **obj** – The object to match against.

`add`(*match*, *handler*)

Register a handler with the Router.

Parameters

- **match** – The first argument passed to the `match()` method when checking against this handler.

- **handler** – A callable or `Route` instance that will handle matching calls. If not a Route instance, will be wrapped in one.

match (*match*, *obj*)
Check for a match.

This method implements the routing logic of the Router. Handlers are registered with a `match` argument, which will be passed to this method when checking against that handler. When the Router is called with a context and an object, it will iterate over its list of registered handlers, passing the corresponding `match` argument and the object to this method once for each, until a match is found. If this method returns a `dict`, it signifies that the object matched against the current handler, and the context is updated with the returned dict. To signify a non-match, this method returns `None`, and iteration continues.

Note: This method is unimplemented in the base class. See `potpy.wsgi.MethodRouter.match()` for a concrete example.

Parameters

- **match** – The `match` argument corresponding to a handler registered with `add()`.
- **obj** – The object to match against.

Returns A `dict` or `None`.

5.3 `potpy.template` – Template module

5.3.1 Module Contents

class `potpy.template.`**`Template`**(*template*, ***type_converters*)
A simple string template class.

Allows you to match against a string, extracting a dictionary of template parameters, with optional parameter type conversion. An example template:

```
>>> t = Template('Hello my name is {name}!')
```

Using the `match()` method, you can extract information from a string:

```
>>> t.match('Hello my name is David!')
{'name': 'David'}

>>> t.match('This string does not match.')
```

Parameter type conversion allows you to coerce parameters to Python types:

```
>>> t = Template('The answer is {answer}', answer=int)
>>> t.match('The answer is 42')
{'answer': 42}
```

You can also specify a regex in your parameter spec to further refine matches:

```
>>> t = Template('/posts/{post_id:\d+}', post_id=int)
>>> t.match('/posts/37')
{'post_id': 37}
>>> t.match('/posts/foo')
```

The reverse of matching is filling. Use the `fill()` method to insert information into your template string:

```
>>> t = Template('The answer is {answer}')
>>> t.fill(answer=42)
'The answer is 42'
```

fill(***kwargs*)
Fill a template string with the given parameters.

```
>>> Template('The answer is {answer}').fill(answer=42)
'The answer is 42'
```

match(*string*)
Match a string against the template.

If the string matches the template, return a dict mapping template parameter names to converted values, otherwise return `None`.

```
>>> t = Template('Hello my name is {name}!')
>>> t.match('Hello my name is David!')
{'name': 'David'}
>>> t.match('This string does not match.')
```

5.4 potpy.wsgi – WSGI module

This module provides classes for creating WSGI (**PEP 333**) applications.

For a simple example, see `examples/wsgi.py`. For a more complete example, see `examples/todo`.

5.4.1 Module Contents

class potpy.wsgi.**PathRouter**(**routes*)
Bases: potpy.router.Router

Route by URL/path.

Utilizes the `Template` class to capture path parameters, adding them to the `Context`. For example, you might define a route with a path template: `/posts/{slug}` – which would match the path `/posts/my-post`, adding `{'slug': 'my-post'}` to the context:

```
>>> from potpy.context import Context
>>> from pprint import pprint
>>> handler = lambda: None  # just a bogus handler
>>> router = PathRouter(('/posts/{slug}', handler))
>>> ctx = Context(path_info='/posts/my-post')
>>> ctx.inject(router)
>>> pprint(dict(ctx))
{'path_info': '/posts/my-post', 'slug': 'my-post'}
```

Routes can also be named, allowing reverse path lookup and filling of path parameters. See `reverse()` for details.

add([*name*], *template*, *handler*)
Add a path template and handler.

Parameters

- **name** – Optional. If specified, allows reverse path lookup with `reverse()`.

- **template** – A string or `Template` instance used to match paths against. Strings will be wrapped in a Template instance.
- **handler** – A callable or `Route` instance which will handle calls for the given path. See `potpy.router.Router.add()` for details.

reverse(*name*, ***kwargs*)

Look up a path by name and fill in the provided parameters.

Example:

```
>>> handler = lambda: None  # just a bogus handler
>>> router = PathRouter(('post', '/posts/{slug}', handler))
>>> router.reverse('post', slug='my-post')
'/posts/my-post'
```

match(*template*, *path_info*)

Check for a path match.

Parameters

- **template** – A `Template` object to match against.
- **path_info** – The path to check for a match.

Returns The template parameters extracted from the path, or `None` if the path does not match the template.

Example:

```
>>> from potpy.template import Template
>>> template = Template('/posts/{slug}')
>>> PathRouter().match(template, '/posts/my-post')
{'slug': 'my-post'}
```

class potpy.wsgi.**MethodRouter**(**routes*)

Bases: `potpy.router.Router`

Route by request method.

```
>>> from potpy.context import Context
>>> handler1 = lambda request_method: (1, request_method.lower())
>>> handler2 = lambda request_method: (2, request_method.lower())
>>> router = MethodRouter(
...     ('POST', handler1),          # can specify a single method
...     (('GET', 'HEAD'), handler2)  # or a tuple of methods
... )
>>> Context(request_method='GET').inject(router)
(2, 'get')
>>> Context(request_method='POST').inject(router)
(1, 'post')
```

exception MethodNotAllowed(*allowed_methods*, *request_method*)

Bases: `potpy.router.NoRoute`

Raised instead of `potpy.router.Router.NoRoute` when no handler matches the given method.

Has an `allowed_methods` attribute which is a list of the methods handled by this router.

MethodRouter.**match**(*methods*, *request_method*)

Check for a method match.

Parameters

- **methods** – A method or tuple of methods to match against.
- **request_method** – The method to check for a match.

Returns An empty `dict` in the case of a match, or `None` if there is no matching handler for the given method.

Example:

```
>>> MethodRouter().match(('GET', 'HEAD'), 'HEAD')
{}
>>> MethodRouter().match('POST', 'DELETE')
```

class `potpy.wsgi.`**App**(*router*, *default_context=None*)

Wrap a potpy router in a WSGI application.

Use this with `PathRouter` and `MethodRouter` to implement a full-featured HTTP request routing system. Return a WSGI app from the last handler in the route, and it will be called with `environ` and `start_response`.

If no route matches, a *404 Not Found* response will be generated. If using a MethodRouter, and the request method doesn't match, a *405 Method Not Allowed* response will be generated. Also responds to HTTP `OPTIONS` requests.

Calls the provided router with a context containing `environ`, `path_info`, and `request_method` fields, and any fields from the optional `default_context` argument.

Parameters

- **router** – The router to call in response to WSGI requests.
- **default_context** – Optional. A `dict`-like mapping of extra fields to add to the context for each request.

Example:

```
>>> def my_app(environ, start_response):
...     start_response('200 OK', [('Content-type', 'text/plain')])
...     return ['Hello, world!']
...
>>> def handler(request):
...     # do something with the request
...     return my_app
...
>>> class Request(object):
...     def __init__(self, environ):
...         pass    # wrap environ in a custom request object
...
>>> app = App(
...     PathRouter(('/hello', lambda: my_app)),
...     {'request': Request}    # add a Request object to context
... )
>>> app({
...     'PATH_INFO': '/hello',
...     'REQUEST_METHOD': 'GET',
... }, lambda status, headers: None)    # bogus start_response
['Hello, world!']
```

__call__(*environ*, *start_response*)

Call the router as a WSGI app.

Constructs a `Context` object with `environ`, `path_info`, and `request_method` (extracted from the environ), and any fields supplied in `self.default_context`.

Calls the result of the router call as a WSGI app.

5.5 potpy.configparser – Config Parser module

Construct a WSGI router from a configuration file.

A configuration file consists of lines specifying URLs, request methods, and handlers, allowing construction of `PathRouter` and `MethodRouter` instances using a hierarchical syntax.

At the top level, you specify URLs with optional names and parameter type converters. See the `Template` class documentation for the converter and URL specification format.

```
foo /foo/{foo_id:\d+} (foo_id: int):
    ...
```

Following each URL is a list of handlers, one on each line. Handlers may also specify a name (in parentheses), in which case the result of the handler is added to the routing context under that name.

```
read_foo (foo)
save_foo
```

It is also possible to specify request method handlers using `* METHOD:` blocks. Adjacent method blocks are combined into a single `MethodRouter` instance.

```
* GET, HEAD:
    show_foo
* POST:
    edit_foo
```

Exception handlers can be specified for a given handler by ending the handler line with a colon (`:`) and listing exception types and handlers on the following lines.

```
read_foo (foo):
    ValidationError, BadFooError: show_foo_errors
    IOError: show_system_errors
```

Complete Example:

```
index /:
    * GET, HEAD:
        views.index
article /{article_id:\d+} (article_id: int):
    * GET, HEAD:
        views.show_article
    * POST:
        auth.require_user (user)
        views.edit_article:
            views.InvalidArticleError: views.show_article_errors
admin /admin/:
    auth.require_admin (user)  # run this regardless of request_method
    * GET, HEAD:
        views.admin_console
```

5.5.1 Module Contents

potpy.configparser.**parse_config**(*lines*, *module=None*)

Parse a config file.

Names referenced within the config file are found within the calling scope. For example:

```
>>> from potpy.configparser import parse_config
>>> class foo:
...     @staticmethod
...     def bar():
...         pass
...
>>> config = '''
... /foo:
...     foo.bar
... '''
>>> router = parse_config(config.splitlines())
```

would find the `bar` method of the `foo` class, because `foo` is in the same scope as the call to parse_config.

Parameters

- **lines** – An iterable of configuration lines (an open file object will do).
- **module** – Optional. If provided and not None, look for referenced names within this object instead of the calling module.

`potpy.configparser.`**`load_config`**(*name='urls.conf'*)

Load a config from a resource file.

The resource is found using pkg_resources.resource_stream(), relative to the calling module.

See `parse_config()` for config file details.

Parameters **name** – The name of the resource, relative to the calling module.

CHAPTER

SIX

INDICES AND TABLES

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

PYTHON MODULE INDEX

p