# Rhetoric Documentation

***Release 0.2.0***

**Maxim Avanov**

January 28, 2015

Contents

Status: **Beta, Unstable API**.

Naive implementation of Pyramid-like routes for Django projects.

CHAPTER 1

# Why it is worth your while

There's a great article on why Pyramid routing subsystem is so convenient for web developers - Pyramid view configuration: Let me count the ways.

As a person who uses Pyramid as a foundation for his pet-projects, and Django - at work, I (the author) had a good opportunity to compare two different approaches to routing configuration provided by these frameworks. And I totally agree with the key points of the article - Pyramid routes are more flexible and convenient for developers writing RESTful services.

The lack of flexibility of standard Django url dispatcher motivated me to create this project. I hope it will be useful for you, and if you liked the idea behind Rhetoric URL Dispatcher, please consider Pyramid Web Framework for one of your future projects.

CHAPTER 2

# Project premises

- Rhetoric components try to follow corresponding Pyramid components whenever possible.
- Integration with django applications shall be transparent to existing code whenever possible.
- Performance of Rhetoric URL Dispatcher is worse than of the one of Pyramid, due to naivety of the implementation and limitations imposed by the compatibility with Django API.

CHAPTER 3

# Installation

Rhetoric is available as a PyPI package:

```
$ pip install Rhetoric
```

The package shall be compatible with Python2.7, and Python3.3 or higher.

CHAPTER 4

# Integration with Django

1. Replace `django.middleware.csrf.CsrfViewMiddleware` with `rhetoric.middleware.CsrfProtectedVie` in your project's `MIDDLEWARE_CLASSES`:

```
# somewhere in a project_name.settings module

MIDDLEWARE_CLASSES = [
    # ...
    'rhetoric.middleware.CsrfProtectedViewDispatchMiddleware',
    #'django.middleware.csrf.CsrfViewMiddleware',
    # ...
]
```

2. Inside the project's root urlconf (usually `project_name.urls`):

```
from django.conf.urls import patterns, include, url
# ... other imports ...
from rhetoric import Configurator

# ... various definitions ...

urlpatterns = patterns('',
    # ... a number of standard django url definitions ...
)

# Rhetorical routing
# ------------------
config = Configurator()
config.add_route('test.new.routes', '/test/new/routes/{param:[a-z]+}')
config.scan(ignore=[
    # do not scan test modules included into the project tree
    re.compile('^.*[.]?tests[.]?.*$').match,
    # do not scan settings modules
    re.compile('^project_name.settings[_]?[_a-z09]*$').match,
])
urlpatterns.extend(config.django_urls())
```

3. Register views:

```
# project_name.some_app.some_module

from rhetoric import view_config


```

```
@view_config(route_name="test.new.routes", renderer='json')
def view_get(request, param):
    return {
        'Hello': param
    }

@view_config(route_name="test.new.routes", renderer='json', request_method='POST')
def view_post(request, param):
    return {
        'Hello': 'POST'
    }
```

4. From this point you can request `/test/new/routes/<param>` with different methods.

CHAPTER 5

# Route Pattern Syntax

**Note:** This section is copied from Pyramid Docs, since Rhetoric provides the same pattern matching functionality.

The *pattern* used in route configuration may start with a slash character. If the pattern does not start with a slash character, an implicit slash will be prepended to it at matching time. For example, the following patterns are equivalent:

```
{foo}/bar/baz
```

and:

```
/{foo}/bar/baz
```

A pattern segment (an individual item between `/` characters in the pattern) may either be a literal string (e.g. `foo`) *or* it may be a replacement marker (e.g. `{foo}`) or a certain combination of both. A replacement marker does not need to be preceded by a `/` character.

A replacement marker is in the format `{name}`, where this means "accept any characters up to the next slash character and use this as the input parameter for a view callable.

A replacement marker in a pattern must begin with an uppercase or lowercase ASCII letter or an underscore, and can be composed only of uppercase or lowercase ASCII letters, underscores, and numbers. For example: `a`, `a_b`, `_b`, and `b9` are all valid replacement marker names, but `0a` is not.

A matchdict is the dictionary representing the dynamic parts extracted from a URL based on the routing pattern. It is available as `request.matchdict`. For example, the following pattern defines one literal segment (`foo`) and two replacement markers (`baz`, and `bar`):

```
foo/{baz}/{bar}
```

The above pattern will match these URLs, generating the following matchdicts:

```
foo/1/2        -> {'baz':u'1', 'bar':u'2'}
foo/abc/def    -> {'baz':u'abc', 'bar':u'def'}
```

It will not match the following patterns however:

```
foo/1/2/        -> No match (trailing slash)
bar/abc/def     -> First segment literal mismatch
```

Replacement markers can optionally specify a regular expression which will be used to decide whether a path segment should match the marker. To specify that a replacement marker should match only a specific set of characters as defined by a regular expression, you must use a slightly extended form of replacement marker syntax. Within braces, the replacement marker name must be followed by a colon, then directly thereafter, the regular expression. The *default* regular expression associated with a replacement marker `[^/]+` matches one or more characters which are not a slash. For example, under the hood, the replacement marker `{foo}` can more verbosely be spelled as `{foo:[^/]+}`. You

can change this to be an arbitrary regular expression to match an arbitrary sequence of characters, such as `{foo:\d+}` to match only digits.

It is possible to use two replacement markers without any literal characters between them, for instance `/{foo}{bar}`. However, this would be a nonsensical pattern without specifying a custom regular expression to restrict what each marker captures.

Segments must contain at least one character in order to match a segment replacement marker. For example, for the URL `/abc/`:

- `/abc/{foo}` will not match.
- `/{foo}/` will match.

CHAPTER 6

# View Configuration Parameters

**Note:** This section is partly copied from the Pyramid documentation, since Rhetoric provides almost the same functionality.

## 6.1 Non-Predicate Arguments

`renderer`

## 6.2 Predicate Arguments

`route_name`

`request_method`

`api_version`

New in version 0.1.7.

Available patterns:

CHAPTER 7

# Renderers

**Note:** This section is copied from the Pyramid Renderers documentation, since Rhetoric provides almost the same rendering functionality.

## 7.1 Built-in renderers

### 7.1.1 `string`: String Renderer

The `string` renderer is a renderer which renders a view callable result to a string. If a view callable returns a non-Response object, and the `string` renderer is associated in that view's configuration, the result will be to run the object through the Python `str` function to generate a string.

### 7.1.2 `json`: JSON Renderer

The `json` renderer renders view callable results to *JSON*. By default, it passes the return value through the `django.core.serializers.json.DjangoJSONEncoder`, and wraps the result in a response object. It also sets the response content-type to `application/json`.

Here's an example of a view that returns a dictionary. Since the `json` renderer is specified in the configuration for this view, the view will render the returned dictionary to a JSON serialization:

```
from rhetoric import view_config

@view_config(renderer='json')
def hello_world(request):
    return {'content':'Hello!'}
```

The body of the response returned by such a view will be a string representing the JSON serialization of the return value:

```
{"content": "Hello!"}
```

### 7.1.3 `.html`: Django Template Renderer

The `.html` template renderer renders views using the standard Django template language. When used, the view must return a HttpResponse object or a Python *dictionary*. The dictionary items will then be used as the template context objects.

## 7.2 Varying Attributes of Rendered Responses

**Note:** This section is partly copied from the Pyramid Renderers documentation, since Rhetoric provides almost the same API.

New in version 0.1.8.

Before a response constructed by a *renderer* is returned to `Django`, several attributes of the request are examined which have the potential to influence response behavior.

View callables that don't directly return a response should use the API of the `django.http.HttpResponse` attribute available as `request.response` during their execution, to influence associated response behavior.

For example, if you need to change the response status from within a view callable that uses a renderer, assign the `status_code` attribute to the `response` attribute of the request before returning a result:

```
from rhetoric import view_config

@view_config(name='dashboard', renderer='dashboard.html')
def myview(request):
    request.response.status_code = 404
    return {'URL': request.get_full_path()}
```

Note that mutations of `request.response` in views which return a HttpResponse object directly will have no effect unless the response object returned *is* `request.response`. For example, the following example calls `request.response.set_cookie`, but this call will have no effect, because a different Response object is returned.

```
from django.http import HttpResponse

def view(request):
    request.response.set_cookie('abc', '123') # this has no effect
    return HttpResponse('OK') # because we're returning a different response
```

If you mutate `request.response` and you'd like the mutations to have an effect, you must return `request.response`:

```
def view(request):
    request.response.set_cookie('abc', '123')
    return request.response
```

## 7.3 Request properties

`request.json_body` - http://docs.pylonsproject.org/projects/pyramid/en/latest/api/request.html#pyramid.request.Request.json_bod

CHAPTER 8

# Predicates

http://docs.pylonsproject.org/docs/pyramid/en/latest/api/config.html#pyramid.config.Configurator.add_view_predicate

CHAPTER 9

# @view_defaults Class Decorator

**Note:** This section is copied from Pyramid Docs, since Rhetoric provides the same functionality.

New in version 0.1.7.

If you use a class as a view, you can use the `rhetoric.view.view_defaults` class decorator on the class to provide defaults to the view configuration information used by every `@view_config` decorator that decorates a method of that class.

For instance, if you've got a class that has methods that represent "REST actions", all which are mapped to the same route, but different request methods, instead of this:

```
from rhetoric import view_config

class RESTView(object):
    def __init__(self, request, *args, **kw):
        self.request = request

    @view_config(route_name='rest', request_method='GET', renderer='json')
    def get(self):
        return {'method': 'GET'}

    @view_config(route_name='rest', request_method='POST', renderer='json')
    def post(self):
        return {'method': 'POST'}

    @view_config(route_name='rest', request_method='DELETE', renderer='json')
    def delete(self):
        return {'method': 'DELETE'}
```

You can do this:

```
from rhetoric import view_config
from rhetoric import view_defaults

@view_defaults(route_name='rest', renderer='json')
class RESTView(object):
    def __init__(self, request, *args, **kw):
        self.request = request

    @view_config(request_method='GET')
    def get(self):
        return {'method': 'GET'}

```

```
13      @view_config(request_method='POST')
14      def post(self):
15          return {'method': 'POST'}
16
17      @view_config(request_method='DELETE')
18      def delete(self):
19          return {'method': 'DELETE'}
```

In the above example, we were able to take the `route_name='rest'` and `renderer='json'` arguments out of the call to each individual `@view_config` statement, because we used a `@view_defaults` class decorator to provide the argument as a default to each view method it possessed.

Arguments passed to `@view_config` will override any default passed to `@view_defaults`.

CHAPTER 10

# ADT

ADT stands for Algebraic Data Type.

```
# -------------------------
# project/payments/models.py
# -------------------------
from rhetoric.adt import adt


# Declare a new ADT
class PaymentMethod(adt):
    # Define variants in a form of VARIANT_NAME = variant_value
    PAYPAL = 'paypal'
    CHEQUE = 'cheque'
    DATACASH = 'bank_transfer'
    ## uncomment the following variant and you will get a configuration error like:
    ##     "Case payment_processor of PaymentMethod is not exhaustive.
    ##      Here is the variant that is not matched: GOOGLE_CHECKOUT"
    ## You will have to implement a payment processor case (see below)
    ## for the GOOGLE_CHECKOUT variant in order to fix the error.
    #GOOGLE_CHECKOUT = 'google_checkout'


# -------------------------
# project/payments/logic.py
# -------------------------
from project.payments.models import PaymentMethod

@PaymentMethod.PAYPAL('payment_processor')
def process_paypal():
    pass

@PaymentMethod.CHEQUE('payment_processor')
def process_cheque():
    pass

@PaymentMethod.DATACASH('payment_processor')
def process_datacash():
    pass

# ----------------------------------------
# Here's the essence of ADT Consistency Check
# ----------------------------------------
## - Uncomment the following definition and you will get a configuration error like:
```

```
## -    "Variant DATACASH of PaymentMethod is already bound to the case payment_processor: process_da
## -
## - You cannot bind variants twice within one case.
##
#@PaymentMethod.DATACASH('payment_processor')
#def process_datacash_error():
#    pass

## - Uncomment the following definition and you will get a standard AttributeError:
## -    "AttributeError: type object 'PaymentMethod' has no attribute 'AMAZON'"
## -
## - You will have to add the AMAZON case to the PaymentMethod ADT in order to fix the error.
##
#@PaymentMethod.AMAZON('payment_processor')
#def process_amazon():
#    pass

## - Uncomment the following definition and you will get a configuration error like:
## -     "Case withdraw_form of PaymentMethod is not exhaustive.
## -      Here is the variant that is not matched: CHEQUE."
## -
## - You will have to implement withdraw forms for all other variants - CHEQUE, DATACASH
## - in order to fix the error.
##
#@PaymentMethod.PAYPAL('withdraw_form')
#class PaypalWitdrawForm(object):
#    pass
#

# ------------------------------------------------------
# Here's the essence of ADT from developer's perspective
# (note the absence of conditional statements such as
#  if:/elif:/elif:/.../else: raise NotImplementedError()
# )
# ------------------------------------------------------

# ----------------------------
# project/payments/__init__.py
# ----------------------------
from project.payments.models import PaymentMethod

def includeme(config):
    RULES = {
        'engine': PaymentMethod
    }
    # The {engine} placeholder will be replaced with the (?:paypal|cheque|bank_transfer) regex.
    # Note that here we use the same ADT object, that was previously used for defining
    # cases payment_processor and withdraw_form.
    config.add_route('payments.withdraw', '/payments/withdraw/{engine}', rules=RULES)

# -------------------------
# project/payments/views.py
# -------------------------
from rhetoric.view import view_config, view_defaults


@view_defaults(route_name='payments.withdraw', renderer='json')
class PaymentsHandler(object)
```

```
    def __init__(self, request, engine):
        self.request = request
        self.engine = engine
        # Note that we will ALWAYS have a proper match here, because this handler
        # will be reached with only correct HTTP requests
        # (i.e. engine value is one of the variant values of PaymentMethod).
        self.payment_strategy = PaymentMethod.match(engine)

    @view_config(request_method='GET', renderer='payments/withdraw_form.html')
    def show_withdraw_form(self):
        # Here, ``payment_strategy.withdraw_form`` is one of case implementations
        # that we defined above with @PaymentMethod(VARIANT, 'withdraw_form').
        # It always points to the relevant implementation!
        form = self.payment_strategy.withdraw_form
        # Render html form
        return {'form': form}

    @view_config(request_method)
    def process_payment(request_method='POST'):
        # Here, ``payment_strategy.payment_processor`` is one of case implementations
        # that we defined above with @PaymentMethod(VARIANT, 'payment_processor').
        # It always points to the relevant implementation!
        processor = self.payment_strategy.payment_processor
        processor()
        # Render json response
        return {'ok': True, 'message': 'Success.'}
```

CHAPTER 11

# Sources

Rhetoric is licensed under the MIT License.

We use GitHub as a primary code repository - https://github.com/avanov/Rhetoric

CHAPTER 12

# Authors

Rhetoric package was created by Maxim Avanov.

- GitHub profile.
- Google+ profile.

CHAPTER 13

# Changelog

- 0.2.0
  - Ported custom predicates
  - Removed support for the `api_version` predicate.
- 0.1.13
  - Depend on Venusian 1.0 and higher.
  - Allow re-assignment of the same ADT case implementations on subsequent venusian scans.
- 0.1.9
  - Added support for the `request.json_body` property.
- 0.1.8
  - Added support for the `request.response` API.
- 0.1.7
  - Added support for the `api_version` predicate.
  - Added the `view_defaults` decorator.
- 0.1.5
  - Feature: added support for `decorator` argument of view_config.
- 0.1.4
  - Feature: added support for custom renderers.
- 0.1.2
  - [Bugfix #2]: resolved race condition in `rhetoric.view.ViewCallback`.
  - [API]: `rhetoric.middleware.UrlResolverMiddleware` was renamed to `rhetoric.middleware.CsrfProtectedViewDispatchMiddleware`.
  - [Django integration]: `rhetoric.middleware.CsrfProtectedViewDispatchMiddleware` should now completely substitute `django.middleware.csrf.CsrfViewMiddleware` in `MIDDLEWARE_CLASSES`.
- 0.1.0 - initial PyPI release. Early development, unstable API.

CHAPTER 14

# Indices and tables

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