pyramid$_{duh}$

Release 0.1.2

October 23, 2016

Contents

This is just a collection of utilities that I found myself putting into *every single* pyramid project I made. So now they're all in one place.

Code lives here: https://github.com/stevearc/pyramid_duh

CHAPTER 1

User Guide

1.1 Request Parameters

There are two provided utilities for accessing request parameters. The first is the `request.param()` method. You can use this method by including `pyramid_duh` in your app (which comes with some *other things*), or if you only want the `param()` method you can include `pyramid_duh.params`:

```
config.include('pyramid_duh')
```

Or in the config file:

```
pyramid.includes =
    pyramid_duh
```

Here is an example use case:

```
def register_user(request):
    username = request.param('username')
    password = request.param('password')
    birthdate = request.param('birthdate', type=date)
    metadata = request.param('metadata', {}, type=dict)
    # insert into database
```

Note that you can pass in default values and perform type conversion. This will handle both form-encoded data and application/json. If a required argument is missing, it will raise a 400. For greater detail, see the function docs at *param()*.

1.1.1 Argify

Let's make the above example sexier:

```
from pyramid_duh import argify

@argify(birthdate=date, metadata=dict)
def register_user(request, username, password, birthdate, metadata=None):
    # insert into database
```

Again, pretty intuitive. If any types are non-unicode, specify them in the *@argify()* decorator. Positional arguments are required; keyword arguments are optional. It even supports the value validation of `request.param()`:

```
from pyramid_duh import argify
```

```
def is_natural_number(num):
    return isinstance(num, int) and num > 0

@argify(age=(int, is_natural_number))
def set_age(request, username, age):
    # Set user age
```

It also makes unit tests nicer:

```
def test_my_view(self):
    request = DummyRequest()
    ret = my_view(request, 'dsa', 'conspiracytheory', date(1989, 4, 1))
```

Note: If you're only using `@argify`, you don't need to include `pyramid_duh`.

1.1.2 Custom Parameter Types

You're now using argument sugar and you're loving it. But you're hungry for more. You want to auto-convert to your own super-special `Unicorn` data type. Well who doesn't?

Here are the POST parameters:

```
{
    username: "stevearc",
    pet: {
        "name": "Sparklelord",
        "sparkly": true,
        "cuddly": true
    }
}
```

And here is the code to parse that mess:

```
class Unicorn(object):
    def __init__(self, name, sparkly, cuddly):
        self.name = name
        self.sparkly = sparkly
        self.cuddly = cuddly

    @classmethod
    def __from_json__(cls, data):
        return cls(**data)


@argify(pet=Unicorn)
def set_user_pet(request, username, pet):
    # Set user pet
```

The `__from_json__` method can be a `classmethod` or a `staticmethod`, and the signature must be either `(arg)` or `(request, arg)`.

Note: I'm using `@argify`, but this also works with `request.param()`.

You can also pass in a factory function as the type:

```
class Unicorn(object):
    def __init__(self, name, sparkly, cuddly):
        self.name = name
        self.sparkly = sparkly
        self.cuddly = cuddly

def make_unicorn(data):
    return Unicorn(**data)

@argify(pet=make_unicorn)
def set_user_pet(request, username, pet):
    # Set user pet
```

If you're running into import dependency hell, you can use a dotted path for the type:

```
@argify(pet='mypkg.models.make_unicorn')
def set_user_pet(request, username, pet):
    # Set user pet
```

1.1.3 Multi-Parameter Types

You can define custom types that will consume multiple request parameters. Let's look at a new set of POST parameters;

```
{
    name: "Sparklelord",
    secret: "Radical",
}
```

Let's say you want to pass up these parameters as login credentials. You would like to fetch the named Unicorn from the database and use that in your view. What would you call that argument? `unicorn` would make sense, but there aren't any parameters named `unicorn`, so how would you inject a parameter that is generated from multiple request parameters? All you need to do is take your type factory function and decorate it with `@argify` as well.

```
@argify
def fetch_unicorn(request, name, secret):
    return request.db.query_for_unicorn(name, secret)

@argify(unicorn=fetch_unicorn)
def make_rainbows(request, unicorn):
    # Make some fukkin' rainbows
```

You'll notice here that we're injecting a field named `unicorn`, which *doesn't exist* in the POST parameters. You can decorate factory methods or the `__from_json__` magic method on type classes.

This particular functionality is kind of magic, and as such I would not recommend using it frequently because it obfuscates your code. This was really made with one thing in mind: user authentication. This is a great way to both authenticate a user and inject the User model into your view with minimal code duplication.

1.2 Add Slash

You have a view. It lies at `http://example.com/path/to/resource/`. But for some reason people keep going to `http://example.com/path/to/resource`. And it messes up relative asset paths. Well, pyramid's solution is a little janky. They define a 404 handler that *always* attempts to add a slash to *any* view that wasn't found. It works, but it's global and you wouldn't know about the behavior just from looking at the view callable. So do this:

```
from pyramid_duh import addslash

@addslash
def my_view(request):
    # serve my resource
```

Easy peasy lemon squeezy.

1.3 Traversal

These are a couple templates for traversal tree nodes that I found myself reusing everywhere.

1.3.1 ISmartLookupResource

This is useful if you have nested resources in your tree, like `/user/1234/post/9876`. You can have a `UserResource` context in your path that has a `user` attribute, and a `PostResource` context that has a `post` attribute. As long as your final context inherits from `ISmartLookupResource`, it can access both the `user` and the `post` directly.

```
@view_config(context=PostResource)
def get_user_post(context, request):
    if context.user.is_cool():
        return context.post
```

This is also useful because it means you don't have to pass the request object down your tree heirarchy. You can just attach it to the root and your nodes will be able to access it.

1.3.2 IStaticResource

Resource for static paths:

```
class MyResource(IStaticResource):
    subobjects = {
        'foo': foo_factory,
        'bar': bar_factory,
    }
```

This does what you think it does. But it prevents you from forgetting to set the `__parent__` and `__name__` attributes on the child. Because that produces terrible and subtle bugs.

1.3.3 IModelResource

Template for retrieving assets from a SQLAlchemy connection. Here's an example:

```
class UserResource(IModelResource):
    __model__ = User
    __modelname__ = 'user'

@view_config(context=UserResource)
def get_user(context, request):
    return context.user
```

This can be customized quite a bit, so look at the docstrings on *IModelResource* for more info.

1.3.4 Where is They?

Just import them

```
from pyramid_duh import ISmartLookupResource, IStaticResource, IModelResource
```

1.4 Subpath Predicate

One problem with pyramid's traversal mechanism is that it doesn't allow you to set view predicates on the subpath. If you aren't already intimately familiar with the details of resource lookup via traversal, here are the docs.

So we've got the `context`, which is the last found resource. The `name`, which is the first url segment that had no new context, and then the `subpath`, which is all path components after the `name`.

To enforce a subpath matching, pass in a list or tuple as the vew predicate:

```
@view_config(context=MyCtxt, name='foobar', subpath=())
def my_view(request):
    # do things
```

Assuming that `MyCtxt` maps to `/mything`, this view will match `/mything/foobar` and `/mything/foobar/` only. No subpath allowed. Here is the format for matching a single subpath:

```
@view_config(context=MyCtxt, name='foobar', subpath=('post', '*'))
def my_view(request):
    id = request.subpath[0]
    # do things
```

You can name the subpaths and access them by name:

```
@view_config(context=MyCtxt, name='foobar', subpath=('post', 'id/*'))
def my_view(request):
    id = request.named_subpaths['id']
    # do things
```

And there are flags you can pass in that allow, among other things, PCRE matching:

```
@view_config(context=MyCtxt, name='foobar', subpath=('type/(post|tweet)/r', 'id/*'))
def my_view(request):
    item_type = request.named_subpaths['type']
    id = request.named_subpaths['id']
    # do things
```

Check the docs on *`SubpathPredicate`* for all of the formats, and *`match()`* for details on match flags.

1.4.1 Including

You can use this predicate by including `pyramid_duh` in your app (which comes with some *other things*), or if you only want the predicate you can include `pyramid_duh.view`:

```
config.include('pyramid_duh')
```

Or in the config file:

```
pyramid.includes =
    pyramid_duh
```

1.5 Settings

`pyramid.settings` has all the useful method for converting to non-string data structures. It has `asbool`, `aslist`, ...actually that's it. We're missing one.

```
users =
    dsa = conspiracytheory
    president_skroob = 12345
```

```
def includeme(config):
    settings = config.get_settings()
    users = asdict(settings['users'])
```

Short and sweet.

1.6 Changelog

1.6.1 0.1.2

- Bug fix: Fix potential timezone issue when converting unix time to datetime

1.6.2 0.1.1

- Bug fix: IStaticResource fails to look up self.request if nested 2-deep
- Bug fix: Name collisions with version_helper.py
- Bug fix: Subpath glob matching always respects case
- Feature: @argify works on view classes
- Feature: @argify can inject types that consume multiple parameters
- Feature: Parameter types can be a dotted path

1.6.3 0.1.0

- Package released into the wild

CHAPTER 2

API Reference

2.1 pyramid_duh package

2.1.1 Submodules

pyramid_duh.auth module

Utilities for auth

class pyramid_duh.auth.**MixedAuthenticationPolicy**(**policies*)

Bases: object

Auth policy that is backed by multiple other auth policies

Checks authentication against each contained policy in order. The first one to return a non-None userid is used. Principals are merged.

add_policy(*policy*)

Add another authentication policy

authenticated_userid(*request*)

Return the authenticated userid or None if no authenticated userid can be found. This method of the policy should ensure that a record exists in whatever persistent store is used related to the user (the user should not have been deleted); if a record associated with the current id does not exist in a persistent store, it should return None.

effective_principals(*request*)

Return a sequence representing the effective principals including the userid and any groups belonged to by the current user, including 'system' groups such as pyramid.security.Everyone and pyramid.security.Authenticated.

forget(*request*)

Return a set of headers suitable for 'forgetting' the current user on subsequent requests.

remember(*request*, *principal*, ***kw*)

Return a set of headers suitable for 'remembering' the principal named principal when set in a response. An individual authentication policy and its consumers can decide on the composition and meaning of **kw.

unauthenticated_userid(*request*)

Return the *unauthenticated* userid. This method performs the same duty as authenticated_userid but is permitted to return the userid based only on data present in the request; it needn't (and shouldn't) check any persistent store to ensure that the user record related to the request userid exists.

pyramid_duh.auth.**includeme**(*config*)
Configure the app

pyramid_duh.compat module

pyramid_duh.params module

Utilities for request parameters

pyramid_duh.params.**argify**(**args*, ***type_kwargs*)
Request decorator for automagically passing in request parameters.

Notes

Here is a sample use case:

```
@argify(foo=dict, ts=datetime)
def handle_request(request, foo, ts, bar='baz'):
    # do request handling
```

No special type is required for strings:

```
@argify
def handle_request(request, foo, bar='baz'):
    # do request handling (both 'foo' and 'bar' are strings)
```

If any positional arguments are missing, it will raise a `HTTPBadRequest` exception. If any keyword arguments are missing, it will simply use whatever the default value is.

Note that unit tests should be unaffected by this decorator. This is valid:

```
@argify
def myview(request, var1, var2='foo'):
    return 'bar'

class TestReq(unittest.TestCase):
    def test_my_request(self):
        request = pyramid.testing.DummyRequest()
        retval = myview(request, 5, var2='foobar')
        self.assertEqual(retval, 'bar')
```

pyramid_duh.params.**includeme**(*config*)
Add parameter utilities

pyramid_duh.params.**is_request**(*obj*)
Check if an object looks like a request

pyramid_duh.params.**param**(*request*, *name*, *default=<object object>*, *type=None*, *validate=None*)
Access a parameter and perform type conversion.

Parameters **request** : `Request`

name : str

The name of the parameter to retrieve

default : object, optional

The default value to use if none is found

type : object, optional

The type to convert the argument to. All python primitives are supported, as well as `date` and `datetime`. You may also pass in a factory function or an object that has a static `__from_json__` method.

validate : callable, optional

Callable test to validate parameter value

Returns **arg** : object

Raises **exc** : `HTTPBadRequest`

If a parameter is requested that does not exist and no default was provided

pyramid_duh.route module

Utilities for traversal

class `pyramid_duh.route.`**IModelResource**(*model=None*)
Bases: *pyramid_duh.route.ISmartLookupResource*

Resource base class for wrapping models in a sqlalchemy database

Notes

Requires any parent node to set the 'request' attribute

create_model(*name*)
Override this if you wish to allow 'PUT' request to create a model

db
Access the SQLAlchemy session on the request

Override this if your session is named something other than 'db'

get_model(*name*)
Retrieve a model from the database

Override this for custom queries

class `pyramid_duh.route.`**ISmartLookupResource**
Bases: `object`

Resource base class that allows hierarchical lookup of attributes

Potential use case: /user/1234/post/5678

At the /user/1234 point in traversal you can set a 'user' attribute on the resource. At the 'post/5678' point in traversal you can set a 'post' attribute on *that* resource. Then the request can access both of them from the context directly:

```
def get_user_post(context, request):
    user = context.user
    if user.is_cool():
        return context.post
```

class `pyramid_duh.route.`**IStaticResource**
Bases: *pyramid_duh.route.ISmartLookupResource*

Simple resource base class for static-mapping of paths

subobjects = {}

pyramid_duh.settings module

Utilities for parsing settings

pyramid_duh.settings.**asdict**(*setting*, *value_type=<function <lambda>>*)

Parses config values from .ini file and returns a dictionary

Parameters **setting** : str

The setting from the config.ini file

value_type : callable

Run this function on the values of the dict

Returns **data** : dict

pyramid_duh.view module

Utilities for view configuration

class pyramid_duh.view.**SubpathPredicate**(*paths*, *config*)

Bases: object

Generate a custom predicate that matches subpaths

Parameters ***paths** : list

List of match specs.

Notes

A match spec may take one of three forms:

```
'glob'
'name/glob'
'name/glob/flags'
```

The name is optional, but if you wish to specify flags then you have to include the leading slash:

```
# A match spec with flags and no name
'/foo.*/r'
```

The names will be accessible from the `request.named_subpaths` attribute.

```
@view_config(context=Root, name='simple', subpath=('package/*', 'version/*/?'))
def simple(request)
    pkg = request.named_subpaths['package']
    version = request.named_subpaths.get('version')
    request.response.body = '<h1>%s</h1>' % package
    if version is not None:
        request.response.body += '<h4>version: %s</h4>' % version
    return request.response
```

See `match()` for more information on match flags‘

phash()

Display name

text()
Display name

pyramid_duh.view.**addslash**(*fxn*)
View decorator that adds a trailing slash

Notes

Usage:

```
@view_config(context=MyCtxt, renderer='json')
@addslash
def do_view(request):
    return 'cool data'
```

pyramid_duh.view.**includeme**(*config*)
Add the custom view predicates

pyramid_duh.view.**match**(*pattern*, *path*, *flags*)
Check if a pattern matches a path

Parameters **pattern** : str

Glob or PCRE

path : str or None

The path to check, or None if no path

flags : {'r', 'i', 'a', '?'}

Special match flags. These may be combined (e.g. 'ri?'). See the notes for an explanation of the different values.

Returns **match** : bool or SRE_Match

A boolean indicating the match status, or the regex match object if there was a successful PCRE match.

Notes

Flag	Description
r	Match using PCRE (default glob)
i	Case-insensitive match (must be used with 'r')
a	ASCII-only match (must be used with 'r', python 3 only)
?	Path is optional (return True if path is None)

2.1.2 Module contents

pyramid_duh

pyramid_duh.**includeme**(*config*)
Add request methods

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

A

C

D

E

F

G

I

M

P

R

S

T

U