TraversalKit Documentation

Release 0.3.1

Dmitry Vakhrushev

Sep 03, 2017

Contents:

The library provides tools to build resource tree for applications that use traversal routing. It has been developed to be used with Pyramid web application framework, however it does not depend on it and can be used within any application.

It helps implement resource tree hierarchy in a simple declarative way:

```
>>> from traversalkit import Resource, DEC_ID

>>> class Root(Resource):
...     """ Tree root """

>>> @Root.mount('users')
... class Users(Resource):
...     """ Users collection """

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     """ User resource """

>>> @Root.mount('posts')
... @User.mount('posts')
... class Posts(Resource):
...     """ Posts collection """

>>> @Posts.mount_set(DEC_ID, metaname='post_id')
... class Post(Resource):
...     """ Post resource """

>>> for route in Root.routes():
...     print(route)
<Route: />
<Route: /posts/>
<Route: /posts/{post_id}/>
<Route: /users/>
<Route: /users/{user_id}/>
<Route: /users/{user_id}/posts/>
<Route: /users/{user_id}/posts/{post_id}/>
```

These resources comply Pyramid traversal interface and Pyramid location awareness interface.

```
>>> root = Root()
>>> user = root['users']['1']
>>> user
<User: /users/1/>
>>> user.__name__
'1'
>>> user.__parent__
<Users: /users/>
>>> user['posts']
<Posts: /users/1/posts/>
>>> user['documents']  # DOCTEST: +ellipsis
Traceback (most recent call last):
...
KeyError: ('documents', '/users/1/')
```

CHAPTER 1

Changes

0.3.1

- Fixed typos of `README.rst` and `CHANGES.rst`.

0.3

- Added support of conditional routes. See module *traversalkit.condition*.
- Added support of resource tree introspection by *traversalkit.resource.Resource.routes()*.
- Added resource URI into raising errors to make them more informative.
- Added support of disengageable resource cache. See class *traversalkit.cache.Cache*.

0.2

- Added method *traversalkit.resource.Resource.get()*.

0.1

Initial release.

CHAPTER 2

Internals

This section contains complete documentation of internal modules that is generated from doc-strings.

traversalkit.resource

Resource

class traversalkit.resource.**Resource**(*name=''*, *parent=None*, *payload=None*, *node=None*)

Base class of resource.

__nodeclass__

Class of route nodes. Links to *traversalkit.route.Node*.

__routeclass__

Class of route. Links to *traversalkit.route.Route*.

__cacheclass__

Class of cache. Links to *traversalkit.cache.Cache*.

__not_exist__

Exception class or list of ones, that should be treated as a signal that resource does not exist.

These exceptions will be replaced by `KeyError` within methods *__getitem__()*, *get()*, and *node()*.

```
>>> import re
>>> from traversalkit import Resource

>>> class Files(Resource):
...     ''' Files collection '''

>>> @Files.mount_set(re.compile(r'^[\w\d\.\_]+$'),
...                  metaname='filename')
... class File(Resource):
```

```
...     ''' File resource '''
...     __not_exist__ = IOError
...     #               ^^^^^^^
...     def on_init(self, payload):
...         with open(self.__name__) as f:
...             self.content = f.read()

>>> files = Files()
>>> files['nonexistent_file.txt']  # DOCTEST: +ellipsis
Traceback (most recent call last):
...
KeyError: ('nonexistent_file.txt', '/')
```

__name__
Resource name, which has been passed to parent's *__getitem__()* or *get()* method.

__parent__
Link to a parent resource. It is actually a property, which stores weak reference to the parent.

__cache__
Cache of child resources. It is used by *__getitem__()* and *get()* methods. Instance of *__cacheclass__*.

__node__
Route node, which has been used to create this resource. Instance of *__nodeclass__*.

__route__
Route, which has been used to create this resource. Instance of *__routeclass__*.

uri
URI of the resource.

classmethod mount (*name*, *class_=None*, *complies=None*, ***kw*)
Mounts single named child resource.

Parameters

- **name** (*str*) – Name of the child resource.
- **class** (Resource) – Child resource class.
- **complies** (Condition) – Condition of the route. See examples of *traversalkit.condition.Under* and *traversalkit.condition.Recursion* for details.

Returns Unmodified class_.

The method can be used as a decorator.

```
>>> from traversalkit import Resource

>>> class Root(Resource):
...     ''' Site root '''

>>> @Root.mount('users')
... class Users(Resource):
...     ''' Collection of users '''

>>> root = Root()
>>> root['users']
<Users: /users/>
```

classmethod mount_set (*pattern*, *class_=None*, *metaname=None*, *complies=None*, ***kw*)
Mounts set of child resources.

Parameters

- **pattern** (*regex*) – Regular expression to match child name.
- **class** (Resource) – Child resource class.
- **metaname** (*str*) – Name of the route. It is used by *node()*.
- **complies** (Condition) – Condition of the route. See examples of *traversalkit.condition.Under* and *traversalkit.condition.Recursion* for details.

Returns Unmodified class_.

The method can be used as a decorator.

```
>>> from traversalkit import Resource, DEC_ID

>>> class Users(Resource):
...     ''' Collection of users '''

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     ''' User resource '''

>>> users = Users()
>>> users['1']
<User: /1/>
>>> users['john']  # DOCTEST: +ellipsis
Traceback (most recent call last):
...
KeyError: ('john', '/')
```

classmethod routes ()
Iterates over all routes available at the current resource.

The method is useful for resource tree introspection and documentation.

Returns Iterator over available routes, see *traversalkit.route.Route*.

```
>>> from traversalkit import Resource, DEC_ID

>>> class Root(Resource):
...     ''' Site root '''

>>> @Root.mount('users')
... class Users(Resource):
...     ''' Collection of users '''

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     ''' User resource '''

>>> for route in Root.routes():
...     print(route)
<Route: />
<Route: /users/>
<Route: /users/{user_id}/>
```

on_init (*payload*)
Initialization callback.

Derived classes should override this method instead of `__init__()`.

Parameters **payload** – Some additional data, that can be passed into initialization routine through *get()* or *node()*. If resource is created by *__getitem__()*, the parameter will be `None`.

__getitem__ (*name*)
Returns child resource by its name.

Child resource class should be mounted using *mount()* or *mount_set()* methods.

The method uses cache. Therefore it won't create child resource, if it's already created.

Parameters **name** (*str*) – Resource name.

Returns Child resource.

Return type *Resource*

Raises **KeyError** – If name does not match any route, or current route does not comply condition.

```
>>> from traversalkit import Resource, DEC_ID

>>> class Root(Resource):
...     ''' Site root '''

>>> @Root.mount('users')
... class Users(Resource):
...     ''' Collection of users '''

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     ''' User resource '''

>>> root = Root()
>>> root['users']
<Users: /users/>
>>> root['users']['1']
<User: /users/1/>
```

get (*name*, *payload=None*)
Returns child resource by its name.

Child resource class should be mounted using *mount()* or *mount_set()* methods.

The method uses cache. Therefore it won't create child resource, if it's already created.

Parameters

- **name** (*str*) – Resource name.
- **payload** – Optional resource payload.

Returns Child resource.

Return type *Resource*

Raises **KeyError** – If name does not match any route, or current route does not comply condition.

```
>>> from traversalkit import Resource, DEC_ID

>>> class Root(Resource):
...     ''' Site root '''

>>> @Root.mount('users')
... class Users(Resource):
...     ''' Collection of users '''

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     ''' User resource '''
...     def on_init(self, payload):
...         if payload is not None:
...             self.id = payload['id']
...             self.name = payload['name']

>>> root = Root()
>>> root['users']
<Users: /users/>
>>> user = root['users'].get('1', {'id': 1, 'name': 'John'})
>>> user
<User: /users/1/>
>>> user.name
'John'
>>> user.id
1
```

node(*name*)

Returns context manager of named route node.

Parameters **name** (*string*) – Name of the route node, i.e. `name` parameter of *`mount()`* or `metaname` parameter of *`mount_set()`*.

Raises **KeyError** – If node does not exist or current route does not comply condition.

The context manager can be used to create multiple child resources, passing route validation only once. It has the following signature:

```
def create_child(name, payload=None):
```

Parameters

- **name** (*string*) – Name of the resource.
- **payload** – Optional resource payload.

Returns Child resource.

Return type *Resource*

```
>>> from traversalkit import Resource, DEC_ID

>>> class Users(Resource):
...     ''' Collection of users '''
...     def index(self):
...         # Let's imagine these data come from DB
...         # and there is a lot of records.
...         users = [{'id': 1, 'name': 'Jonh'},
```

```
...                     {'id': 2, 'name': 'Jane'}]
...             with self.node('user_id') as create_child:
...             #    ^^^^^^^^^^^^^^^^^^^
...             #    here we pass route validation
...                 for user in users:
...                     yield create_child(str(user['id']), user)
...                     #     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
...                     #     and here create multiple resources

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     ''' User resource '''
...     def on_init(self, payload):
...         if payload is not None:
...             self.id = payload['id']
...             self.name = payload['name']

>>> users = Users()
>>> john, jane = users.index()
>>> john.name
'Jonh'
>>> jane.name
'Jane'
>>> users['1'] is john
True
>>> users['2'] is jane
True
```

lineage()

Returns iterator over resource parents.

Lineage chain includes current resource.

```
>>> from traversalkit import Resource, DEC_ID

>>> class Root(Resource):
...     ''' Site root '''

>>> @Root.mount('users')
... class Users(Resource):
...     ''' Collection of users '''

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     ''' User resource '''

>>> root = Root()
>>> list(root['users']['1'].lineage())
[<User: /users/1/>, <Users: /users/>, <Root: />]
```

parent(*name=None*, *cls=None*)

Searches particular parent in the resource lineage.

Parameters

- **name** (*str*) – Optional parent name.
- **cls** (Resource, *str*) – Optional class or class name of parent.

Returns Parent resource or None.

```
>>> from traversalkit import Resource, DEC_ID

>>> class Root(Resource):
...     ''' Site root '''

>>> @Root.mount('users')
... class Users(Resource):
...     ''' Collection of users '''

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     ''' User resource '''

>>> root = Root()
>>> user = root['users']['1']
>>> user.parent('users')
<Users: /users/>
>>> user.parent(cls='Root')
<Root: />
>>> user.parent(cls=Root)
<Root: />
>>> user.parent()
<Users: /users/>
```

child(*class_*, *name*, *payload=None*)

> **Warning:** Deprecated in favor of `node()`.

Creates child resource from given class and name.

ResourceMeta

class traversalkit.resource.**ResourceMeta**(*class_name*, *bases*, *attrs*)
Resource metaclass

traversalkit.ids

The module provides most common ID patterns.

traversalkit.ids.**ANY_ID**
Wildcard matching. Matches everything: `.*`

traversalkit.ids.**DEC_ID**
Matches decimal numbers only: `^[\d]+$`

traversalkit.ids.**HEX_ID**
Matches hexadecimal numbers only: `^[a-f\d]+$`

traversalkit.ids.**TEXT_ID**
Matches single word: `^[\w\-]+$`

traversalkit.route

The module provides route descriptors.

The following classes should not be instantiated directly. They are used within *traversalkit.resource.Resource* to handle routes.

Node

class traversalkit.route.**Node**(*class_*, *name=None*, *pattern=None*, *metaname=None*, *complies=None*)

Route node descriptor.

Parameters

- **class** (Resource) – Resource class of the node.
- **name** (*str*) – Name of the node. Optional.
- **pattern** (*regex*) – Pattern of node name. Optional.
- **metaname** (*str*) – Metaname of node. Optional.
- **complies** (Condition) – Condition that route should complie. Optional.

class_
Resource class of the node. Should be a subclass of *traversalkit.resource.Resource*

name
Name of the node. It is specified, when the node is created by *traversalkit.resource.Resource.mount()*.

pattern
Pattern of the node name. It is specified, when the node is created by *traversalkit.resource.Resource.mount_set()*.

metaname
Metaname of the node. It is specified, when the node is created by *traversalkit.resource.Resource.mount_set()*.

type
Type of the node.

If *name* is defined, the `type` will be `"single"`, i.e. the node has been created using *traversalkit.resource.Resource.mount()*.

If *name* is not defined, the `type` will be `"set"`, i.e. the node has been created using *traversalkit.resource.Resource.mount_set()*.

complies(*route*)
Checks whether the route complies node's condition.

If the node has been created without `complies` parameter, this method will always return `True`.

If the node has been created with `complies` parameter, this method will run `compiles(route + self)` (i.e. passes the route concatenated with the node itself to the condition) and return the result. See *traversalkit.condition* for details.

Parameters **route** (Route) – Route to test.

Returns Result of the test.

Return type bool

__str__()
String representation of the node.

It is mostly useful for documentation purposes. There are three possible representations:

```
>>> import re

>>> # Node describes single named resource
>>> node = Node(object, name='foo')
>>> str(node)
'foo'

>>> # Node describes anonymous set of resources
>>> node = Node(object, pattern=re.compile('.*'))
>>> str(node)
'{.*}'

>>> # Node describes named set of resources
>>> node = Node(object, pattern=re.compile('.*'),
...             metaname='foo')
>>> str(node)
'{foo}'
```

Route

class traversalkit.route.**Route**(**nodes*)
Route descriptor.

In general, it is just a immutable sequence of nodes (see *Node*) with some syntactic surgar.

Parameters ***nodes** (Node) – Nodes of the route.

```
>>> import re
>>> route = Route(Node(object, name=''))
>>> route
<Route: />
>>> len(route)
1
>>> route.uri
'/'

>>> route += Node(object, name='foo')
>>> route
<Route: /foo/>
>>> len(route)
2
>>> route.uri
'/foo/'

>>> route += [
...     Node(object, pattern=re.compile(r'.*')),
...     Node(object, pattern=re.compile(r'.*'), metaname='bar'),
... ]
>>> route
<Route: /foo/{.*}/{bar}/>
>>> len(route)
4
```

```
>>> route.uri
'/foo/{.*}/{bar}/'
```

traversalkit.condition

The module provides condition descriptors of routes.

Condition

class traversalkit.condition.**Condition**
Base class for condition.

The class provides implementation of some syntactic sugar:

```
>>> Condition() & Condition()
And(Condition(), Condition())

>>> Condition() | Condition()
Or(Condition(), Condition())

>>> ~Condition()
Not(Condition())
```

Derived class should only override *__call__()* method.

__call__(*route*)
Test route against the condition.

Parameters **route** (Route) – Route to test.

Returns The result of test, i.e. True if given route complies the condition, or False otherwise.

Return type bool

Raises **NotImplementedError** – If not overridden.

And

class traversalkit.condition.**And**(*left*, *rigth*)
Utility class to concatenate wrapped conditions by logical AND.

It does not have to be used directly. Use & operator instead:

```
>>> Condition() & Condition()
And(Condition(), Condition())
```

Parameters

- **left** (Condition) – Left condition.
- **right** (Condition) – Right condition.

Or

class traversalkit.condition.**Or**(*left*, *rigth*)
Utility class to concatenate wrapped conditions by logical OR.

It does not have to be used directly. Use | operator instead:

```
>>> Condition() | Condition()
Or(Condition(), Condition())
```

Parameters

- **left** (Condition) – Left condition.
- **right** (Condition) – Right condition.

Not

class traversalkit.condition.**Not**(*condition*)
Utility class to negate wrapped condition.

It does not have to be used directly. Use ~ operator instead:

```
>>> ~Condition()
Not(Condition())
```

Parameters **condition** (Condition) – Condition to negate.

Under

class traversalkit.condition.**Under**(**parents*)
Test that route contains specified parents.

Parameters ***parents** (Resource, *str*) – List of resource classes or resource names.

The following example describes simple blog with two sections of posts: published ones and drafts. Published post can contain comments section, but draft cannot. This restriction is done using *Under* condition negated by *Not* (i.e. ~ operator).

```
>>> from traversalkit import Resource, DEC_ID

>>> class Blog(Resource):
...     ''' Blog root resource '''

>>> @Blog.mount('posts')    # Published posts can contain comments
... @Blog.mount('drafts')   # Draft posts cannot contain comments
... class Posts(Resource):
...     ''' Blog posts collection '''

>>> @Posts.mount_set(DEC_ID, metaname='post_id')
... class Post(Resource):
...     ''' Blog post '''

>>> @Post.mount('comments', complies=~Under('drafts'))
... class Comments(Resource):  #     ^^^^^^^^^^^^^^^^
...     ''' Comments collection '''
```

```
>>> blog = Blog()
>>> blog['posts']['1']['comments']
<Comments: /posts/1/comments/>

>>> blog['drafts']['2']['comments']  # DOCTEST: +ellipsis
Traceback (most recent call last):
...
KeyError: ('comments', '/drafts/2/')

>>> for route in Blog.routes():
...     print(route)
<Route: />
<Route: /drafts/>
<Route: /drafts/{post_id}/>
<Route: /posts/>
<Route: /posts/{post_id}/>
<Route: /posts/{post_id}/comments/>
```

Recursion

class `traversalkit.condition.`**`Recursion`**(*maxdepth*)

Test that route does not go deeper than specified recursion depth.

Parameters **`maxdepth`** (*int*) – Maximum recursion depth.

The following example describes categories tree. Where collection resource `Categories` contains item resource `Category`, which again contains collection `Categories`. To restrict this endless route *Recursion* condition is used with maximum depth of 2.

```
>>> from traversalkit import Resource, DEC_ID

>>> class Categories(Resource):
...     ''' Categories collection '''

>>> @Categories.mount_set(DEC_ID, metaname='category_id')
... class Category(Resource):
...     ''' Category resource '''

>>> Category.mount(
...     'categories', Categories,
...     complies=Recursion(maxdepth=2),
...     #        ^^^^^^^^^^^^^^^^^^^^
... )   # DOCTEST: +ellipsis
<class ...

>>> tree = Categories()
>>> tree['1']['categories']['2']
<Category: /1/categories/2/>

>>> tree['1']['categories']['2']['categories']  # DOCTEST: +ellipsis
Traceback (most recent call last):
...
KeyError: ('categories', '/1/categories/2/')

>>> for route in Categories.routes():
...     print(route)
```

```
<Route: />
<Route: /{category_id}/>
<Route: /{category_id}/categories/>
<Route: /{category_id}/categories/{category_id}/>
```

traversalkit.cache

Cache

class traversalkit.cache.**Cache**(**args*, ***kw*)

Resource cache.

Provides regular mutable mapping interface with additional `readonly` method. The `readonly` returns a context manager, that can be used to temporary freeze cache state.

```
>>> cache = Cache()
>>> cache['x'] = 1
>>> cache['y'] = 2
>>> cache == {'x': 1, 'y': 2}
True

>>> with cache.readonly():
...     cache['x'] = 3
...     del cache['y']

>>> cache == {'x': 1, 'y': 2}
True
```

It is useful when you do not want to cache child resources:

```
>>> from traversalkit import Resource, DEC_ID

>>> class Users(Resource):
...     ''' Collection of users '''

>>> @Users.mount_set(DEC_ID, metaname='user_id')
... class User(Resource):
...     ''' User resource '''

>>> users = Users()
>>> with users.__cache__.readonly():
...     user_1 = users['1']

>>> user_2 = users['2']
>>> users['2'] is user_2
True
>>> users['1'] is user_1
False
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

U