Pyoko

Release 0.6.2

January 22, 2016

Contents

Contents:

CHAPTER 1

pyoko package

1.1 Subpackages

1.1.1 pyoko.db package

Submodules

pyoko.db.queryset module

this module contains a base class for other db access classes

class pyoko.db.queryset.**QuerySet**(**conf*)
Bases: object

QuerySet is a lazy data access layer for Riak.

__deepcopy__(*memo=None*)
A deep copy method that doesn't populate caches and shares Riak client and bucket

_clear_bucket()
only for development purposes

_compile_query()
Builds SOLR query and stores it into self.compiled_query

_escape_query(*query*, *escaped=False*)
Escapes query if it's not already escaped.

Parameters

- **query** – Query value.
- **escaped** (*bool*) – expresses if query already escaped or not.

Returns Escaped query value.

_exec_query()
Executes solr query if it hasn't already executed.

Returns Self.

_get()
executes solr query if needed then returns first object according to selected ReturnType (defaults to Model)
:return: pyoko.Model or riak.Object or solr document

_make_model(*data*, *riak_obj=None*)
Creates a model instance with the given data.

Parameters

- **data** – Model data returned from DB.
- **riak_obj** –

Returns pyoko.Model object.

_parse_query_key(*key*, *val*)
Strips query modifier from key and call's the appropriate value modifier.

Parameters

- **key** (*str*) – Query key
- **val** – Query value

Returns Parsed query key and value.

_parse_query_modifier(*modifier*, *query_value*)
Parses query_value according to query_type

Parameters

- **modifier** (*str*) – Type of query. Exact, contains, lte etc.
- **query_value** – Value partition of the query.

Returns Parsed query_value.

_process_params()
Adds default row size if it's not given in the query. Converts param values into unicode strings.

Returns Processed self._solr_params dict.

_save_model(*model=None*)
saves the model instance to riak :return:

_set_bucket(*type*, *name*)
prepares bucket, sets index name :param str type: bucket type :param str name: bucket name :return:

count()
counts by executing solr query with rows=0 parameter :return: number of objects matches to the query :rtype: int

data()
set return type as riak objects instead of pyoko models

distinct_values_of(*field*)

exclude(***filters*)
Applies query filters for excluding matching records from result set.

Parameters ****filters** – Query filters as keyword arguments.

Returns Self. Queryset object.

Examples

```
>>> Person.objects.exclude(age=None)
>>> Person.objects.filter(name__startswith='jo').exclude(age__lte=16)
```

filter(***filters*)
Applies given query filters.

Parameters ****filters** – Query filters as keyword arguments.

Returns Self. Queryset object.

Examples

```
>>> Person.objects.filter(name='John') # same as .filter(name__exact='John')
>>> Person.objects.filter(age__gte=16, name__startswith='jo')
>>> # Assume u1 and u2 as related model instances.
>>> Person.objects.filter(work_unit__in=[u1, u2], name__startswith='jo')
```

get(*key=None*, ***kwargs*)
Ensures that only one result is returned from DB and raises an exception otherwise. Can work in 3 different way.

•If no argument is given, only does "ensuring about one and only object" job.

•If key given as only argument, retrieves the object from DB.

•if query filters given, implicitly calls filter() method.

Raises `MultipleObjectsReturned` – If there is more than one (1) record is returned.

get_or_create(*defaults=None*, ***kwargs*)
Looks up an object with the given kwargs, creating a new one if necessary.

Parameters

- **defaults** (*dict*) – Used when we create a new object. Must map to fields of the model.
- ****kwargs** – Used both for filtering and new object creation.

Returns A tuple of (object, created), where created is a boolean variable specifies whether the object was newly created or not.

Example

In the following example, *code* and *name* fields are used to query the DB.

```
obj, is_new = Permission.objects.get_or_create({'description': desc},
                                               code=code, name=name)
```

{description: desc} dict is just for new creations. If we can't find any records by filtering on *code* and *name*, then we create a new object by using all of the inputs.

or_filter(***filters*)
Works like "filter" but joins given filters with OR operator.

Parameters ****filters** – Query filters as keyword arguments.

Returns Self. Queryset object.

Example

```
>>> Person.objects.or_filter(age__gte=16, name__startswith='jo')
```

raw(*query*, ***params*)
make a raw query :param str query: solr query :param dict params: solr parameters

search_on(**fields*, ***query*)
Search for query on given fields.

Query modifier can be one of these: # * exact * contains * startswith * endswith * range * lte * gte

Parameters

- ***fields** (*str*) – Field list to be searched on
- ****query** – Search query. While it's implemented as **kwargs we only support one (first) keyword argument.

Returns Self. Queryset object.

Examples

```
>>> Person.objects.search_on('name', 'surname', contains='john')
>>> Person.objects.search_on('name', 'surname', startswith='jo')
```

set_params(***params*)
add/update solr query parameters

solr()
set return type for raw solr docs

class pyoko.db.queryset.**ReturnType**
Bases: enum.Enum

Model = <ReturnType.Model: 3>

Object = <ReturnType.Object: 2>

Solr = <ReturnType.Solr: 1>

pyoko.db.connection module

riak client configuration

pyoko.db.schema_update module

class pyoko.db.schema_update.**FakeContext**
Bases: object

has_permission(*perm*)

class pyoko.db.schema_update.**SchemaUpdater**(*registry*, *bucket_names*, *threads*, *force*)
Bases: object

traverses trough all models, collects fields marked for index or store in solr then creates a solr schema for these fields.

FIELD_TEMPLATE = '<field type="{type}" name="{name}" indexed="{index}" stored="{store}" multiValued="{multi}"

static **apply_schema**(*client*, *force*, *job_pack*)
riak doesn't support schema/index updates (http://git.io/vLOTS)

as a workaround, we create a temporary index, attach it to the bucket, delete the old index/schema, re-create the index with new schema, assign it to bucket, then delete the temporary index.

Parameters

- **new_schema** (*byte*) – compiled schema
- **bucket_name** (*str*) – name of schema, index and bucket.

Returns True or False

Return type bool

compile_schema(*fields*)
joins schema fields with base solr schema

Parameters **fields** (*list[str]*) – field list

Returns compiled schema

Return type byte

create_report()
creates a text report for the human user :return: str

classmethod **get_schema_fields**(*fields*)

Parameters **fields** (*list[(,)]*) – field props tupple list

Return type list[str]

Returns schema fields list

run()

pyoko.db.schema_update.**get_schema_from_solr**(*index_name*)

pyoko.db.schema_update.**wait_for_schema_creation**(*index_name*)

pyoko.db.schema_update.**wait_for_schema_deletion**(*index_name*)

Module contents

1.1.2 pyoko.lib package

Submodules

pyoko.lib.py2map module

tools to convert Python dicts to / from riak Maps

pyoko.lib.utils module

class pyoko.lib.utils.**MyEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)
Bases: json.encoder.JSONEncoder

default(*obj*)

class pyoko.lib.utils.**SimpleChoicesManager**
Bases: object

static get_all(*choices*)

class pyoko.lib.utils.**SimpleRiakFileManager**
Bases: object

store_file(***kw*)

pyoko.lib.utils.**add_to_path**()

pyoko.lib.utils.**getScriptPath**()

pyoko.lib.utils.**get_object_from_path**(*path*)
Import's object from given Python path.

pyoko.lib.utils.**grayed**(**args*)

pyoko.lib.utils.**pprnt**(*input*, *return_data=False*)
Prettier print for nested data

Parameters

- **input** – Input data
- **return_data** (*bool*) – Default False. Print outs if False, returns if True.

Returns None | Pretty formatted text representation of input data.

pyoko.lib.utils.**random_word**(*length*)

pyoko.lib.utils.**to_camel**(*s*)

Parameters **s** (*string*) – under_scored string to be CamelCased

Returns CamelCase version of input

Return type str

pyoko.lib.utils.**un_camel**(*input*)

pyoko.lib.utils.**un_camel_id**(*input*)
uncamel for id fields :param input: :return:

Module contents

1.2 Submodules

1.3 pyoko.conf module

class pyoko.conf.**Settings**
Bases: object

1.4 pyoko.exceptions module

exception pyoko.exceptions.**MultipleObjectsReturned**
Bases: *pyoko.exceptions.PyokoError*

The query returned multiple objects when only one was expected.

exception pyoko.exceptions.**NoSuchObjectError**
Bases: *pyoko.exceptions.PyokoError*

exception pyoko.exceptions.**NotCompatible**
Bases: *pyoko.exceptions.PyokoError*

Incorrect usage of method / function

exception pyoko.exceptions.**ObjectDoesNotExist**
Bases: *pyoko.exceptions.PyokoError*

exception pyoko.exceptions.**PyokoError**
Bases: exceptions.Exception

exception pyoko.exceptions.**ValidationError**
Bases: *pyoko.exceptions.PyokoError*

1.5 pyoko.fields module

class pyoko.fields.**BaseField**(*title=''*, *default=None*, *required=True*, *index=False*, *type=None*, *store=False*, *choices=None*, *order=None*, ***kwargs*)
Bases: object

_load_data(*instance*, *value*)
for some field types (eg:date, datetime) we treat differently to data that came from db and given by user

clean_value(*val*)

creation_counter = 2

default_value = None

validate(*val*)

class pyoko.fields.**Boolean**(*title=''*, *default=None*, *required=True*, *index=False*, *type=None*, *store=False*, *choices=None*, *order=None*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

clean_value(*val*)

solr_type = 'boolean'

class pyoko.fields.**Date**(**args*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

clean_value(*val*)

solr_type = 'date'

class pyoko.fields.**DateTime**(**args*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

clean_value(*val*)

solr_type = 'date'

class pyoko.fields.**File**(**args*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

clean_value(*val*)
val = :param dict val: {"content":"", "name":"", "ext":"", "type":""} :return:

file_manager

solr_type = 'file'

class pyoko.fields.**Float**(*title=''*, *default=None*, *required=True*, *index=False*, *type=None*, *store=False*, *choices=None*, *order=None*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

Numeric field that holds float data.

clean_value(*val*)

solr_type = 'float'

class pyoko.fields.**Id**(**arg*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

clean_value(*val*)

solr_type = 'string'

class pyoko.fields.**Integer**(*title=''*, *default=None*, *required=True*, *index=False*, *type=None*, *store=False*, *choices=None*, *order=None*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

clean_value(*val*)

default_value = 0

solr_type = 'int'

class pyoko.fields.**String**(*title=''*, *default=None*, *required=True*, *index=False*, *type=None*, *store=False*, *choices=None*, *order=None*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

solr_type = 'string'

class pyoko.fields.**Text**(*title=''*, *default=None*, *required=True*, *index=False*, *type=None*, *store=False*, *choices=None*, *order=None*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

Text field.

solr_type = 'text_general'

class pyoko.fields.**TimeStamp**(**args*, ***kwargs*)
Bases: *pyoko.fields.BaseField*

clean_value(*val*)

solr_type = 'long'

1.6 pyoko.listnode module

This module holds the ListNode implementation of Pyoko Models.

ListNode's are used to model ManyToMany relations and other list like data types on a Model.

class pyoko.listnode.**ListNode**(***kwargs*)
Bases: *pyoko.node.Node*

ListNode's are used to store list of field sets. Their DB representation look like list of dicts:

```
class Student(Model):
    class Lectures(ListNode):
        name = field.String()
```

```
            code = field.String(required=False)

    st = Student()
    st.Lectures(name="Math101", code='M1')
    st.Lectures(name="Math102", code='M2')
    st.clean_value()
    {
        'deleted': False,
        'timestamp': None
        'lectures': [
            {'code': 'M1', 'name': 'Math101'},
            {'code': 'M2', 'name': 'Math102'},
        ]
    }
```

Notes

- Currently we disregard the ordering of ListNode items.

__call__(***kwargs*)
Stores created instance in node_stack and returns it's reference to callee

__delattr__
x.__delattr__('name') <==> del x.name

__delitem__(*obj*)
Allow usage of "del" statement on ListNodes with bracket notation.

Parameters **obj** – ListNode item or relation key.

Raises TypeError – If it's called on a ListNode item (intstead of ListNode's itself)

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__()

This works for two different object:

- Main ListNode object
- Items of the ListNode (like instance of a class) which created while iterating on main ListNode object

Returns String representation of object.

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

_generate_instances()
ListNode item generator. Will be used internally by __iter__ and __getitem__

Yields ListNode items (instances)

_get_linked_model_key()
Only one linked model can represent a listnode instance,

Returns The first linked models key if exists otherwise None

_load_data(*data*, *from_db=False*)
Stores the data at self._data, actual object creation done at _generate_instances()

Parameters

- **data** (*list*) – List of dicts.
- **from_db** (*bool*) – Default False. Is this data coming from DB or not.

_make_instance(*node_data*)
Create a ListNode instance from node_data

Parameters **node_data** (*dict*) – Data to create ListNode item.

Returns ListNode item.

add(***kwargs*)
Stores node data without creating an instance of it. This is more efficient if node instance is not required.

Parameters **kwargs** – attributes of the ListNode

clean_value()
Populates json serialization ready data. This is the method used to serialize and store the object data in to DB

Returns List of dicts.

clear()
Clear outs the list node.

Raises `TypeError` – If it's called on a ListNode item (intstead of ListNode's itself)

get_field(*field_name*)

get_humane_value(*name*)
Returns a human readble/meaningful value for the field

Parameters **name** (*str*) – Model field name

Returns Human readble field value

get_link(***kw*)

get_links(***kw*)

get_verbose_name()

remove()
Removes an item from ListNode.

Raises `TypeError` – If it's called on a ListNode item (intstead of ListNode's itself)

Note: Parent object should be explicitly saved.

1.7 pyoko.manage module

command line management interface

class pyoko.manage.**Command**(*manager*)
Bases: object

Command object is a thin wrapper around Python's powerful argparse module. Holds the given command line parameters in self.manager.args

CMD_NAME
name of your command

HELP
help texts starts with "Rl" will be parsed as raw text

PARAMS
A dictionary list with following possible values.

- name: name of parameter
- help: help text for parameter. Parsed as raw if starts with "Rl"
- required: Optional. Set True if this is a required parameter.
- default: Optional. Define a default value for the parameter
- action: 'store_true' see the official argparse documentation for more info

run()
This is where the things are done. You should override this method in your command class.

class pyoko.manage.**CommandRegistry**(*mcs*, *name*, *bases*, *attrs*)
Bases: type

classmethod add_command(*command_model*)

classmethod get_commands()

CommandRegistry.registry [{}]

class pyoko.manage.**DumpData**(*manager*)
Bases: *pyoko.manage.Command*

CHOICES = ('csv', 'json', 'json_tree', 'pretty')

CMD_NAME = 'dump_data'

CSV = 'csv'

HELP = 'Dumps all data to stdout or to given file'

JSON = 'json'

DumpData.PARAMS [{}]

PRETTY = 'pretty'

TREE = 'json_tree'

run()

class pyoko.manage.**FindDuplicateKeys**(*manager*)
Bases: *pyoko.manage.Command*

CMD_NAME = '_find_dups'

HELP = 'finds duplicate keys, to help debugging'

run()

class pyoko.manage.**FlushDB**(*manager*)
Bases: *pyoko.manage.Command*

CMD_NAME = 'flush_model'

HELP = 'REALLY DELETES the contents of buckets'

FlushDB.PARAMS [{}]

run()

class pyoko.manage.**LoadData**(*manager*)
Bases: *pyoko.manage.Command*

Loads previously dumped data into DB.

CHOICES = ('csv', 'json', 'json_tree', 'pretty')

CMD_NAME = 'load_data'

CSV = 'csv'

HELP = 'Reads JSON data from given file and populates models'

JSON = 'json'

LoadData.PARAMS [{}]

PRETTY = 'pretty'

TREE = 'json_tree'

prepare_buckets()
loads buckets to bucket cache. disables the default json encoders if CSV is selected

read_file(*file_path*)

read_json_per_line(*file*)

read_per_line(*file*)

read_whole_file(*file*)

run()

save_obj(*bucket_name*, *key*, *val*)

class pyoko.manage.**ManagementCommands**(*args=None*)
Bases: object

All CLI commands executed by this class. You can create your own commands by extending Command class

parse_args(*args*)

class pyoko.manage.**SchemaUpdate**(*manager*)
Bases: *pyoko.manage.Command*

CMD_NAME = 'migrate'

HELP = 'Creates/Updates SOLR schemas for given model(s)'

SchemaUpdate.PARAMS [{}]

run()

class pyoko.manage.**Shell**(*manager*)
Bases: *pyoko.manage.Command*

CMD_NAME = 'shell'

HELP = 'Run IPython shell'

Shell.PARAMS [{}]

run()

class pyoko.manage.**SmartFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)
Bases: argparse.HelpFormatter

class pyoko.manage.**TestGetKeys**(*manager*)
Bases: *pyoko.manage.Command*

CMD_NAME = '_test_get_keys'

HELP = 'tests the correctness of the bucket.get_keys()'

run()

1.8 pyoko.model module

class pyoko.model.**LinkProxy**(*link_to*, *one_to_one=False*, *verbose_name=None*, *reverse_name=None*)
Bases: object

Proxy object for "self" referencing model relations .. rubric:: Example

```
class Unit(Model):
    name = field.String("Name")
    parent = LinkProxy('Unit', verbose_name='Upper unit', reverse_name='sub_units')
```

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__

class pyoko.model.**Model**(*context=None*, ***kwargs*)

Bases: *pyoko.node.Node*

This is base class for any model object.

Field instances are used as model attributes to represent values.

```
class Permission(Model):
    name = field.String("Name")
    code = field.String("Code Name")

    def __unicode__(self):
        return "%s %s" % (self.name, self.code)
```

Models may have inner classes to represent ManyToMany relations, inner data nodes or lists.

__delattr__
x.__delattr__('name') <==> del x.name

__eq__(*other*)
Equivalence of two model instance depends on uniformity of their self._data and self.key.

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

_apply_cell_filters(*context*)

Applies the field restrictions based on the return value of the context's "has_permission()" method. Stores them on self._unpermitted_fields.

Returns List of unpermitted fields names.

_handle_changed_fields(*old_data*)
Looks for changed relation fields between new and old data (before/after save). Creates back_link references for updated fields.

Parameters **old_data** – Object's data before save.

_update_new_linked_model(*linked_mdl_ins*, *link*)
Iterates through linked_models of given model instance to match it's "reverse" with given link's "field" values.

clean_value()
generates a json serializable representation of the model data :rtype: dict :return: riak ready python dict

delete()
This method just flags the object as "deleted" and saves it to DB.

exist
Used to check if a relation is exist or a model instance is saved to DB or not.

Returns True if this model instance stored in DB and has a key and False otherwise.

Examples

```
>>> class Student(Model):
>>>     #...
>>>     adviser = Person()
>>>
>>> if student.adviser.exist:
>>>     # do something
```

get_choices_for(*field*)
Get the choices for the given fields.

Parameters **field** (*str*) – Name of field.

Returns List of tuples. [(name, value),...]

get_field(*field_name*)

get_humane_value(*name*)
Returns a human readble/meaningful value for the field

Parameters **name** (*str*) – Model field name

Returns Human readble field value

get_link(***kw*)

get_links(***kw*)

get_unpermitted_fields()
Gives unpermitted fields for current context/user.

Returns List of unpermitted field names.

get_verbose_name()

Returns Verbose name of the model instance

is_in_db()

Deprecated: Use "exist" property instead.

objects
alias of `QuerySet`

post_save()
Called after object save. Can be overriden to do things that should be done after object saved to DB.

pre_save()
Called before object save. Can be overriden to do things that should be done just before object saved to DB.

prnt()
Prints DB data representation of the object.

static **row_level_access**(*context*, *objects*)
If defined, will be called just before query compiling step and it's output summed up to existing query filter.

Can be used to implement context-aware implicit filtering. You can define your query filters in here to enforce row level access control.

Parameters

- **context** – An object that contain required user attributes and permissions.
- **objects** (*Queryset*) – QuerySet object.

Examples

```
>>> return objects.filter(user=context.user)
```

save (*internal=False*)
Save's object to DB.

Do not override this method, use pre_save and post_save methods.

Parameters **internal** (*bool*) – True if called within model. Used to prevent unneccessary calls to pre_save and post_save methods.

Returns Saved model instance.

set_data (*data*, *from_db=False*)
Fills the object's fields with given data dict. Internally calls the self._load_data() method.

Parameters

- **data** (*dict*) – Data to fill object's fields.
- **from_db** (*bool*) – if data coming from db then we will
- **related field type's _load_data method** (*use*) –

Returns Self. Returns objects itself for chainability.

1.9 pyoko.modelmeta module

class pyoko.modelmeta.**ModelMeta** (*mcs*, *name*, *bases*, *attrs*)
Bases: type

Metaclass that process model classes.

static process_attributes_of_node (*attrs*, *node_name*, *class_type*)
prepare the model fields, nodes and relations

Parameters

- **node_name** (*str*) – name of the node we are currently processing
- **attrs** (*dict*) – attribute dict
- **class_type** (*str*) – Type of class. Can be one of these: 'ListNode', 'Model', 'Node'

static process_models (*attrs*, *base_model_class*)
Attach default fields and meta options to models

static process_objects (*kls*)
Applies default Meta properties.

1.10 pyoko.node module

class pyoko.node.**FakeContext**
Bases: object

this fake context object can be used to use ACL limited models from shell

has_permission(*perm*)

class pyoko.node.**LazyModel**(*wrapped*, *verbose_name*)
Bases: Proxy

exist

get_verbose_name()

key = None

verbose_name = None

class pyoko.node.**Node**(***kwargs*)
Bases: object

We store node classes in _nodes[] attribute at ModelMeta, then replace them with their instances at _instantiate_nodes()

Likewise we store linked models in _linked_models[]

Since fields are defined as descriptors, they can access to instance they called from but to access their methods and attributes, we're copying fields themself into self._fields[] attribute. So, we get values of fields from self._field_values[] and access to fields themselves from self._fields[]

_collect_index_fields(*in_multi=False*)
Collects fields which will be indexed.

Parameters **in_multi** (*bool*) – if we are in a ListNode or not

Returns [(field_name, solr_type, is_indexed, is_stored, is_multi]

_instantiate_nodes()
instantiate all nodes

_load_data(*data*, *from_db=False*)
With the data returned from riak: - fills model's fields, nodes and listnodes - instantiates linked model instances

Parameters

- **from_db** (*bool*) – if data coming from db instead of calling self._set_fields_values() we simply use field's _load_data method.
- **data** (*dict*) –

Returns self

_path_of(*prop*)
returns the dotted path of the given model attribute

_set_fields_values(*kwargs*)
Fill the fields of this node

Parameters **kwargs** – Field values

clean_value()
generates a json serializable representation of the model data :rtype: dict :return: riak ready python dict

classmethod **get_field**(*field_name*)

get_humane_value(*name*)
Returns a human readble/meaningful value for the field

Parameters **name** (*str*) – Model field name

Returns Human readble field value

classmethod **get_link**(***kw*)

classmethod **get_links**(***kw*)

get_verbose_name()

1.11 pyoko.registry module

class pyoko.registry.**Registry**
Bases: object

get_apps()

get_base_models()

get_model(*model_name*)

get_models_by_apps()

get_models_of_app(*app_name*)

register_model(*mdl*)

1.12 Module contents

CHAPTER 2

Indices and tables

- genindex
- search

p

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W