mogwai Documentation

Release 0.7.5

Cody Lee

August 07, 2015

Contents

mogwai is an object graph mapper (OGM) designed for Titan Graph Database.

CHAPTER 1

Features

- Straightforward syntax
- Query Support
- Vertex and Edge Support
- Uses RexPro Connection Pooling
- Gevent and Eventlet socket support via RexPro
- Supports Relationship quick-modeling
- Supports custom gremlin scripts and methods
- Doesn't restrict the developer from using direct methods.
- Save Strategies
- Property Validation
- Optionally can utilize factory_boy to generate models.
- Interactive shell available
- Performance monitoring tools available.
- Serialization support for Pickle
- Support for un-modeled properties with dictionary-like access
- Support for imports via groovy and the groovy Script Engine. (*See Example*)
- Tested with Python 2.7, 3.3 and 3.4

CHAPTER 2

Links

- Documentation: http://mogwai.readthedocs.org/
- Official Repository: https://bitbucket.org/wellaware/mogwai.git
- Package: https://pypi.python.org/pypi/mogwai/

mogwai is known to support Python 2.7, 3.3 and 3.4.

CHAPTER 3

Download

PyPI: https://pypi.python.org/pypi/mogwai/

```
$ pip install mogwai
```

Source: https://bitbucket.org/wellaware/mogwai.git

```
$ git clone git://bitbucket.org/wellaware/mogwai.git
$ python setup.py install
```

CHAPTER 4

Contents

4.1 Quick Start

4.1.1 Usage

Note: This section provides a quick summary of mogwai features. A more detailed listing is available in the full documentation.

4.1.2 Setup Connection

You'll need to setup the connection to the graph database. Mogwai handles connection pooling for you, but you should handle load balancing using dedicated equipment.

```
from mogwai.connection import setup

# Without Authentication
setup('localhost')
# With authentication
#setup('localhost', username='rexster', password='rexster')
# With gevent support
#setup('localhost', concurrency='gevent')  # default is Standard Synchronous Python Sockets
# With eventlet support
#setup('localhost', concurrency='eventlet')  # default is Standard Synchronous Python Sockets
```

4.1.3 Defining Models

Models declare a Vertex or Edge model and attributes it should possibly contain.

```
from mogwai.models import Vertex, Edge
from mogwai.properties import String, Integer

_val = 0

def counter():
    global _val
    _val += 1
    return _val
```

```
class TestVertexModel(Vertex):
    element_type = 'test_vertex_model'

    name = String(default='test_vertex')
    test_val = Integer(default=counter)


class TestEdgeModel(Edge):
    label = 'test_edge_model'

    name = String(default='test_edge')
    test_val = Integer(default=counter)
```

4.1.4 Using Models

mogwai models can be utilized in a similar fashion to the way the Django ORM was constructed, but tailored for Graph Databases.

Creating Vertex or Edge

```
my_vertex_1 = TestVertexModel.create(name='my vertex')
my_vertex_2 = TestVertexModel.create(name='my other vertex')
my_edge = TestEdgeModel.create(outV=my_vertex_1, inV=my_vertex_2, name='my edge')
```

Retriveing a Vertex or Edge

```
# Get all the TestVertexModel Vertices
vertices = TestVertexModel.all()
# Get a subset of vertices by titan ID
vertices = TestVertexModel.all([1234, 5678, 9012])
# Get a vertex by titan ID
vertex = TestVertexModel.get(1234)

# Getting all Edges isn't currently supported
# Get a subset of edges by titan IDs
edges = TestEdgeModel.all(['123-UX4', '215-PX3', '95U-32Z'])
# get a single edge by titan ID
edge = TestEdgeModel.get('123-UX4')

# Get edge between two vertices
edge = TestEdgeModel.get_between(outV=my_vertex_1, inV=my_vertex_2)
```

Simple Traversals

Vertex Traversals

```
# Get All Edges from the vertex
edges = my_vertex_1.bothE()
# Get outgoing edges from the vertex
edges = my_vertex_1.outE()
# Get incoming edges to the vertex
```

```
edges = my_Vertex_1.inE()
# Specify an edge type for any edge traversal operation (works for outE, inE, bothE)
## By using models
test_model_edges = my_vertex_1.outE(TestEdgeModel)
## or by using manual labels
test_model_edges = my_vertex_1.outE('test_edge_model')

# Get all Vertices connected to the vertex
vertices = my_vertex_1.bothV()
# Get all vertices who are connected by edge coming into the current vertex (note uni-directed edges
vertices = my_vertex_1.outV()
# Get all vertices who are connected by edge coming from the current vertex (note uni-directed edges
vertices = my_vertex_1.inV()
# Specify an edge type for any edge traversal operation (works for outV, inV, bothV)
## By using models
test_model_vertices = my_vertex_1.outV(TestEdgeModel)
## or by using manual element types
test_model_vertices = my_vertex_1.outV('test_edge_model')
```

Edge Traversals

```
# Get the vertex which is from the outgoing side of the edge
vertex = my_edge.outV()
# Get the vertex which is from the incoming side of the edge
vertex = my_edge.inV()
```

4.2 Reference

4.2.1 Connection

mogwai.connection.**execute_query**(**args*, ***kwargs*)

Execute a raw Gremlin query with the given parameters passed in.

Parameters

- **query** (*str*) – The Gremlin query to be executed
- **params** (*dict*) – Parameters to the Gremlin query
- **connection** (*RexPro(Sync|Gevent|Eventlet)Connection or None*) – The RexPro connection to execute the query with
- **context** – String context data to include with the query for stats logging

Return type dict

mogwai.connection.**generate_spec**()

Generates a titan index and type specification document based on loaded Vertex and Edge models

mogwai.connection.**get_response**(*query*, *params*, *isolate*, *transaction*, *connection*, *connection_pool*)

mogwai.connection.**pop_execute_query_kwargs**(*keyword_arguments*)

pop the optional execute query arguments from arbitrary kwargs; return non-None query kwargs in a dict

mogwai.connection.**setup**(*host*, *graph_name=u'graph'*, *graph_obj_name=u'g'*, *username=u''*, *password=u''*, *metric_reporters=None*, *pool_size=10*, *concurrency=u'sync'*)
Sets up the connection, and instantiates the models

mogwai.connection.**sync_spec**(*filename*, *host*, *graph_name=u'graph'*, *graph_obj_name=u'g'*, *username=u''*, *password=u''*, *dry_run=False*)
Sync the given spec file to mogwai.

Parameters

- **filename** (*str*) – The filename of the spec file
- **host** (*str*) – The host the be synced
- **graph_name** (*str*) – The name of the graph to be synced
- **dry_run** (*boolean*) – Only prints generated Gremlin if True

Returns None

4.2.2 Vertex

class mogwai.models.vertex.**EnumVertexBaseMeta**
This metaclass allows you to access MyVertexModel as if it were an enum. Ex. MyVertexModel.FOO

The values are cached in a dictionary. This is useful if the number of MyVertexModels is small, however it it grows too large, you should be doing it a different way.

This looks for a special (optional) function named *enum_generator* in your model and calls that to generate the ENUM for the model.

There is an additional optional model attribute that can be set *__enum_id_only__* (defaults to True) which dictates whether or not just the Vertex ID is stored, or the whole Vertex in cache.

enums = None

mro() → list
return a type's method resolution order

class mogwai.models.vertex.**Vertex**(***values*)
The Vertex model base class.

The element type is auto-generated from the subclass name, but can optionally be set manually

exception DoesNotExist
Object not found in database

args

message

Vertex.**FACTORY_CLASS = None**

exception Vertex.**MultipleObjectsReturned**
Multiple objects returned on unique key lookup

args

message

exception Vertex.**WrongElementType**
Unique lookup with key corresponding to vertex of different type

args

message

classmethod Vertex.**all**(*ids=[]*, *as_dict=False*, *match_length=True*, **args*, ***kwargs*)
Load all vertices with the given ids from the graph. By default this will return a list of vertices but if as_dict is True then it will return a dictionary containing ids as keys and vertices found as values.

Parameters

- **ids** (*list*) – A list of titan ids
- **as_dict** (*boolean*) – Toggle whether to return a dictionary or list

Return type dict | list

Vertex.**as_dict**()
Returns a map of column names to cleaned values

Return type dict

Vertex.**as_save_params**()
Returns a map of property names to cleaned values containing only the properties which should be persisted on save.

Return type dict

Vertex.**bothE**(**labels*, ***kwargs*)
Return a list of edges both incoming and outgoing from this vertex.

Parameters

- **label** (*str or BaseEdge or None*) – The edge label to be traversed (optional)
- **limit** (*int or None*) – The number of the page to start returning results at
- **offset** (*int or None*) – The maximum number of results to return
- **types** (*list*) – A list of allowed element types

Vertex.**bothV**(**labels*, ***kwargs*)
Return a list of vertices both incoming and outgoing from this vertex.

Parameters

- **label** (*str or BaseEdge or None*) – The edge label to be traversed (optional)
- **limit** (*int or None*) – The number of the page to start returning results at
- **offset** (*int or None*) – The maximum number of results to return
- **types** (*list*) – A list of allowed element types

Vertex.**create**(**args*, ***kwargs*)
Create a new element with the given information.

Vertex.**delete**()
Delete the current vertex from the graph.

Vertex.**delete_inE**(**labels*)
Delete all incoming edges with the given label.

Vertex.**delete_inV**(**labels*)
Delete all incoming vertices connected with edges with the given label.

Vertex.**delete_outE**(**labels*)
Delete all outgoing edges with the given label.

Vertex.**delete_outV**(**labels*)
Delete all outgoing vertices connected with edges with the given label.

Vertex.**deserialize**(*data*)
Deserializes rexpro response into vertex or edge objects

Vertex.**element_type = None**

classmethod Vertex.**find_by_value**(*field*, *value*, *as_dict=False*)
Returns vertices that match the given field/value pair.

Parameters

- **field** (*str*) – The field to search
- **value** (*str*) – The value of the field
- **as_dict** (*boolean*) – Return results as a dictionary

Return type [mogwai.models.Vertex]

classmethod Vertex.**get**(*id*, **args*, ***kwargs*)
Look up vertex by its ID. Raises a DoesNotExist exception if a vertex with the given vid was not found. Raises a MultipleObjectsReturned exception if the vid corresponds to more than one vertex in the graph.

Parameters **id** (*str*) – The ID of the vertex

Return type mogwai.models.Vertex

classmethod Vertex.**get_element_type**()
Returns the element type for this vertex.

@returns: str

Vertex.**get_property_by_name**(*key*)
Get's the db_field_name of a property by key

Parameters **key** (*basestring | str*) – attribute of the model

Return type basestring | str | None

Vertex.**gremlin_path = u'vertex.groovy'**

Vertex.**id**

Vertex.**inE**(**labels*, ***kwargs*)
Return a list of edges with the given label coming into this vertex.

Parameters

- **label** (*str or BaseEdge*) – The edge label to be traversed
- **limit** (*int or None*) – The number of the page to start returning results at
- **offset** (*int or None*) – The maximum number of results to return
- **types** (*list*) – A list of allowed element types

Vertex.**inV**(**labels*, ***kwargs*)
Return a list of vertices reached by traversing the incoming edge with the given label.

Parameters

- **label** (*str or BaseEdge*) – The edge label to be traversed
- **limit** (*int or None*) – The number of the page to start returning results at
- **offset** (*int or None*) – The maximum number of results to return

- **types** (*list*) – A list of allowed element types

Vertex.**items**()

Vertex.**keys**()

Vertex.**outE**(**labels*, ***kwargs*)
Return a list of edges with the given label going out of this vertex.

Parameters

- **label** (*str or BaseEdge*) – The edge label to be traversed
- **limit** (*int or None*) – The number of the page to start returning results at
- **offset** (*int or None*) – The maximum number of results to return
- **types** (*list*) – A list of allowed element types

Vertex.**outV**(**labels*, ***kwargs*)
Return a list of vertices reached by traversing the outgoing edge with the given label.

Parameters

- **labels** (*str or BaseEdge*) – pass in the labels to follow in as positional arguments
- **limit** (*int or None*) – The number of the page to start returning results at
- **offset** (*int or None*) – The maximum number of results to return
- **types** (*list*) – A list of allowed element types

Vertex.**pre_save**()
Pre-save hook which is run before saving an element

Vertex.**pre_update**(***values*)
Override this to perform pre-update validation

Vertex.**query**()

Vertex.**reload**(**args*, ***kwargs*)
Reload the given element from the database.

Vertex.**save**(**args*, ***kwargs*)
Save the current vertex using the configured save strategy, the default save strategy is to re-save all fields every time the object is saved.

Vertex.**translate_db_fields**(*data*)
Translates field names from the database into field names used in our model

this is for cases where we're saving a field under a different name than it's model property

Parameters **data** – dict

Return type dict

Vertex.**update**(***values*)
performs an update of this element with the given values and returns the saved object

Vertex.**validate**()
Cleans and validates the field values

Vertex.**validate_field**(*field_name*, *val*)
Perform the validations associated with the field with the given name on the value passed.

Parameters

- **field_name** (*str*) – The name of property whose validations will be run

- **val** (*mixed*) – The value to be validated

Vertex.**values**()

class mogwai.models.vertex.**VertexMetaClass**
Metaclass for vertices.

mro() → list
return a type's method resolution order

4.2.3 Edge

class mogwai.models.edge.**Edge**(*outV*, *inV*, ***values*)
Base class for all edges.

exception DoesNotExist
Object not found in database

args

message

Edge.**FACTORY_CLASS = None**

exception Edge.**MultipleObjectsReturned**
Multiple objects returned on unique key lookup

args

message

exception Edge.**WrongElementType**
Unique lookup with key corresponding to vertex of different type

args

message

classmethod Edge.**all**(*ids*, *as_dict=False*, **args*, ***kwargs*)
Load all edges with the given edge_ids from the graph. By default this will return a list of edges but if as_dict is True then it will return a dictionary containing edge_ids as keys and edges found as values.

Parameters

- **ids** (*list*) – A list of titan IDs
- **as_dict** (*boolean*) – Toggle whether to return a dictionary or list

Return type dict | list

Edge.**as_dict**()
Returns a map of column names to cleaned values

Return type dict

Edge.**as_save_params**()
Returns a map of property names to cleaned values containing only the properties which should be persisted on save.

Return type dict

classmethod Edge.**create**(*outV*, *inV*, *label=None*, **args*, ***kwargs*)
Create a new edge of the current type coming out of vertex outV and going into vertex inV with the given properties.

Parameters

- **outV** (Vertex) – The vertex the edge is coming out of
- **inV** (Vertex) – The vertex the edge is going into

Edge.**delete**()
Delete the current edge from the graph.

Edge.**deserialize**(*data*)
Deserializes rexpro response into vertex or edge objects

classmethod Edge.**find_by_value**(*field*, *value*, *as_dict=False*)
Returns edges that match the given field/value pair.

Parameters

- **field** (*str*) – The field to search
- **value** (*str*) – The value of the field
- **as_dict** (*boolean*) – Return results as a dictionary

Return type [mogwai.models.Edge]

classmethod Edge.**get**(*id*, **args*, ***kwargs*)
Look up edge by titan assigned ID. Raises a DoesNotExist exception if a edge with the given edge id was not found. Raises a MultipleObjectsReturned exception if the edge_id corresponds to more than one edge in the graph.

Parameters **id** (*str | basestring*) – The titan assigned ID

Return type mogwai.models.Edge

classmethod Edge.**get_between**(*outV*, *inV*, *page_num=None*, *per_page=None*)
Return all the edges with a given label between two vertices.

Parameters

- **outV** (Vertex) – The vertex the edge comes out of.
- **inV** (Vertex) – The vertex the edge goes into.
- **page_num** (*int*) – The page number of the results
- **per_page** (*int*) – The number of results per page

Return type list

classmethod Edge.**get_label**()
Returns the label for this edge.

Return type str

Edge.**get_property_by_name**(*key*)
Get's the db_field_name of a property by key

Parameters **key** (*basestring | str*) – attribute of the model

Return type basestring | str | None

Edge.**gremlin_path = 'edge.groovy'**

Edge.**id**

Edge.**inV**(**args*, ***kwargs*)
Return the vertex that this edge goes into.

Return type *Vertex*

Edge.**items**()

Edge.**keys**()

Edge.**label = None**

Edge.**outV**(**args*, ***kwargs*)
Return the vertex that this edge is coming out of.

Return type *Vertex*

Edge.**pre_save**()
Pre-save hook which is run before saving an element

Edge.**pre_update**(***values*)
Override this to perform pre-update validation

Edge.**reload**(**args*, ***kwargs*)
Reload the given element from the database.

Edge.**save**(**args*, ***kwargs*)
Save this edge to the graph database.

Edge.**translate_db_fields**(*data*)
Translates field names from the database into field names used in our model

this is for cases where we're saving a field under a different name than it's model property

Parameters **data** – dict

Return type dict

Edge.**update**(***values*)
performs an update of this element with the given values and returns the saved object

Edge.**validate**()
Perform validation of this edge raising a ValidationError if any problems are encountered.

Edge.**validate_field**(*field_name*, *val*)
Perform the validations associated with the field with the given name on the value passed.

Parameters

- **field_name** (*str*) – The name of property whose validations will be run
- **val** (*mixed*) – The value to be validated

Edge.**values**()

class mogwai.models.edge.**EdgeMetaClass**
Metaclass for edges.

mro() → list
return a type's method resolution order

4.2.4 Gremlin

class mogwai.gremlin.base.**BaseGremlinMethod**(*path=None*, *method_name=None*, *classmethod=False*, *property=False*, *defaults=None*, *transaction=True*, *imports=None*)
Maps a function in a groovy file to a method on a python class

configure_method(*klass*, *attr_name*, *gremlin_path*)
Sets up the methods internals

Parameters

- **klass** (*object*) – The class object this function is being added to
- **attr_name** (*str*) – The attribute name this function will be added as
- **gremlin_path** (*str*) – The path to the gremlin file containing method

transform_params_to_database(*params*)
Takes a dictionary of parameters and recursively translates them into parameters appropriate for sending over Rexpro.

Parameters **params** (*dict*) – The parameters to be sent to the function

Return type dict

class mogwai.gremlin.base.**GremlinMethod**(*path=None*, *method_name=None*, *classmethod=False*, *property=False*, *defaults=None*, *transaction=True*, *imports=None*)
Gremlin method that returns a graph element

configure_method(*klass*, *attr_name*, *gremlin_path*)
Sets up the methods internals

Parameters

- **klass** (*object*) – The class object this function is being added to
- **attr_name** (*str*) – The attribute name this function will be added as
- **gremlin_path** (*str*) – The path to the gremlin file containing method

transform_params_to_database(*params*)
Takes a dictionary of parameters and recursively translates them into parameters appropriate for sending over Rexpro.

Parameters **params** (*dict*) – The parameters to be sent to the function

Return type dict

class mogwai.gremlin.base.**GremlinTable**(*path=None*, *method_name=None*, *classmethod=False*, *property=False*, *defaults=None*, *transaction=True*, *imports=None*)
Gremlin method that returns a table as its result

configure_method(*klass*, *attr_name*, *gremlin_path*)
Sets up the methods internals

Parameters

- **klass** (*object*) – The class object this function is being added to
- **attr_name** (*str*) – The attribute name this function will be added as
- **gremlin_path** (*str*) – The path to the gremlin file containing method

transform_params_to_database(*params*)
Takes a dictionary of parameters and recursively translates them into parameters appropriate for sending over Rexpro.

Parameters **params** (*dict*) – The parameters to be sent to the function

Return type dict

class mogwai.gremlin.base.**GremlinValue**(*path=None*, *method_name=None*, *classmethod=False*, *property=False*, *defaults=None*, *transaction=True*, *imports=None*)

Gremlin Method that returns one value

configure_method(*klass*, *attr_name*, *gremlin_path*)

Sets up the methods internals

Parameters

- **klass** (*object*) – The class object this function is being added to
- **attr_name** (*str*) – The attribute name this function will be added as
- **gremlin_path** (*str*) – The path to the gremlin file containing method

transform_params_to_database(*params*)

Takes a dictionary of parameters and recursively translates them into parameters appropriate for sending over Rexpro.

Parameters **params** (*dict*) – The parameters to be sent to the function

Return type dict

mogwai.gremlin.groovy.**GroovyFileDef**

alias of GroovyFileDefinition

class mogwai.gremlin.groovy.**GroovyFunction**(*name*, *args*, *body*, *defn*)

args

Alias for field number 1

body

Alias for field number 2

count(*value*) → integer – return number of occurrences of value

defn

Alias for field number 3

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

name

Alias for field number 0

class mogwai.gremlin.groovy.**GroovyFunctionParser**

Given a string containing a single function definition this class will parse the function definition and return information regarding it.

FuncDefn = {{{{{“def” Re:(‘[A-Za-z_]\\w*’)} “(“} Re:(‘[A-Za-z_]\\w*’) [, Re:(‘[A-Za-z_]\\w*’)]...} ”)”} “{“}

FuncName = Re:(‘[A-Za-z_]\\w*’)

KeywordDef = “def”

VarName = Re:(‘[A-Za-z_]\\w*’)

classmethod parse(*data*)

Parse the given function definition and return information regarding the contained definition.

Parameters **data** (*str | basestring*) – The function definition in a string

Return type dict

class mogwai.gremlin.groovy.**GroovyImport**(*comment_list*, *import_strings*, *import_list*)

comment_list
Alias for field number 0

count(*value*) → integer – return number of occurrences of value

import_list
Alias for field number 2

import_strings
Alias for field number 1

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

class mogwai.gremlin.groovy.**GroovyImportParser**
Given a string containing a single import definition this class will parse the import definition and return information regarding it.

CommentVar = comment

ImportDef = Suppress:("import")

ImportDefn = {{{Suppress:("import") Re:('[A-Za-z_.*]*') [. Re:('[A-Za-z_.*]*')]...} Suppress:(";")} [{Suppress:("//")

ImportVarName = Re:('[A-Za-z_.*]*')

OptionalSpace = [" "]

classmethod parse(*data*)
Parse the given import and return information regarding the contained import statement.

Parameters **data** (*str | basestring*) – The import statement in a string

Return type dict

mogwai.gremlin.groovy.**parse**(*filename*)
Parse Groovy code in the given file and return a list of information about each function necessary for usage in queries to database.

Parameters **filename** (*str*) – The file containing groovy code.

Return type list

class mogwai.gremlin.table.**Table**(*gremlin_result*)
A table accepts the results of a GremlinTable in it's constructor. It can be iterated over like a normal list, but within the rows the dictionaries are accessible via .notation

For example:

returns a table of people & my friend edge to them # the edge contains my nickname for that person friends = mogwai.gremlin.GremlinTable()

def get_friends_and_my_nickname(self): result = self.friends() for i in result:

print "{}:{}".format(i.friend_edge.nickname, i.person.name)

next()

class mogwai.gremlin.table.**Row**(*data*)
A row represent a table row, from which it's columns can be accessed like a tuple or dict. Rows are read-only and accept elements or dicts with as initializers as a result of a GremlinTable query. Also the . getattr notation can be used to access elements

Example: row = Row({'person': Friend.create(....), 'myval': 3}) print "{}:{} - {}".format(row.friend_edge.nickname, row.person.name, row.myval)

items()

iteritems()

keys()

next()

values()

4.2.5 Properties

class mogwai.properties.base.**BaseValueManager**(*graph_property*, *value*, *strategy=<class 'mogwai.properties.strategy.SaveAlways'>*)

Value managers are used to manage values pulled from the database and track state changes.

These are useful for save strategies.

changed

Indicates whether or not this value has changed.

Return type bool

deleted

Indicates whether or not this value has been deleted.

Return type bool

delval()

Delete a given value

get_property()

Returns a value-managed property attributes

Return type property

getval()

Return the current value.

previous_value

setval(*val*)

Updates the current value.

param val The new value

class mogwai.properties.base.**GraphProperty**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)

Base class for graph property types

can_delete

data_type = u'Object'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

classmethod get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.BaseValidator object>

value_manager
alias of *BaseValueManager*

class mogwai.properties.properties.**Boolean**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)

Boolean Data property type

can_delete

data_type = 'Boolean'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.BooleanValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**DateTime**(*strict=True*, ***kwargs*)
UTC DateTime Data property type

can_delete

data_type = 'Double'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.DateTimeUTCValidator object>

value_manager
alias of BaseValueManager

class mogwai.properties.properties.**DateTimeNaive**(*strict=True*, ***kwargs*)
DateTime Data property type

can_delete

data_type = 'Double'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.DateTimeValidator object>

value_manager
alias of `BaseValueManager`

class mogwai.properties.properties.**Decimal**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)
Decimal Data property type

can_delete

data_type = u'Object'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.DecimalValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**Dictionary**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)
Dictionary Data property type

can_delete

data_type = 'HashMap'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.DictValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**Double**(***kwargs*)
Double Data property type

can_delete

data_type = 'Double'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.FloatValidator object>

value_manager
alias of `BaseValueManager`

class mogwai.properties.properties.**Email**(**args*, ***kwargs*)
Email Data property type

can_delete

data_type = 'String'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)

validator = <mogwai.properties.validators.EmailValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**Float**(***kwargs*)
Float class for backwards compatability / if you really want to

can_delete

data_type = 'Double'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.FloatValidator object>

value_manager
alias of `BaseValueManager`

class mogwai.properties.properties.**IPV4**(**args*, ***kwargs*)
IPv4 Data property type

can_delete

data_type = 'String'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)

validator = <mogwai.properties.validators.RegexValidator object>

value_manager
alias of `BaseValueManager`

class mogwai.properties.properties.**IPV6**(**args*, ***kwargs*)
IPv6 Data property type

can_delete

data_type = 'String'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)

validator = <mogwai.properties.validators.RegexValidator object>

value_manager
alias of BaseValueManager

class mogwai.properties.properties.**IPV6WithV4**(**args*, ***kwargs*)
IPv6 with Mapped/Translated/Embedded IPv4 Data property type

can_delete

data_type = 'String'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)

validator = <mogwai.properties.validators.RegexValidator object>

value_manager
alias of BaseValueManager

class mogwai.properties.properties.**Integer**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)
Integer Data property type

can_delete

data_type = 'Integer'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.LongValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**List**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)
List Data property type

can_delete

data_type = 'ArrayList'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.ListValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**Long**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)
Long Data property type

can_delete

data_type = 'Long'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.LongValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**PositiveInteger**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)
Positive Integer Data property type

can_delete

data_type = 'Integer'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.PositiveIntegerValidator object>

value_manager
alias of `BaseValueManager`

class mogwai.properties.properties.**PositiveLong**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix=u''*)

Positive Long Data property type

can_delete

data_type = 'Long'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.PositiveIntegerValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**Short**(*description=None*, *primary_key=False*, *index=False*, *index_ext=None*, *db_field=None*, *choices=None*, *default=None*, *required=False*, *save_strategy=<class 'mogwai.properties.strategy.SaveAlways'>*, *unique=None*, *db_field_prefix='u'*)
Short Data property type

can_delete

data_type = 'Short'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.IntegerValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**Slug**(**args*, ***kwargs*)
Slug Data property type

can_delete

data_type = 'String'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)

validator = <mogwai.properties.validators.RegexValidator object>

value_manager
alias of BaseValueManager

class mogwai.properties.properties.**String**(**args*, ***kwargs*)
String/CharField property

can_delete

data_type = 'String'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)

validate(*value*)

validator = <mogwai.properties.validators.StringValidator object>

value_manager
alias of `BaseValueManager`

mogwai.properties.properties.**Text**
alias of *String*

class mogwai.properties.properties.**URL**(**args*, ***kwargs*)
URL Data property type

can_delete

data_type = 'String'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)
Converts python value into database value

Return type Object

to_python(*value*)
Converts data from the database into python values raises a ValidationError if the value can't be converted

Return type Object

validate(*value*)

validator = <mogwai.properties.validators.URLValidator object>

value_manager
alias of `BaseValueManager`

class `mogwai.properties.properties.`**UUID**(*default=<function <lambda>>*, ***kwargs*)
Universally Unique Identifier (UUID) type - UUID4 by default

can_delete

data_type = 'String'

db_field_name
Returns the name of the mogwai name of this graph property

Return type basestring | str

get_default()
Returns the default value for this graph property if one is available.

Return type Object | None

get_save_strategy()
Returns the save strategy attached to this graph property.

Return type Callable

get_value_from_choices(*value*, *choices*)
Returns the key for the choices tuple of tuples

Note if you are using classes, they must implement the __in__ and __eq__ for the logical comparison.

Parameters **value** (*Object*) – The raw value to test if it exists in the valid choices. Could be the key or the value in the dict

Return type Object

has_db_field_prefix
Determines if a field prefix has already been defined.

has_default
Indicates whether or not this graph property has a default value.

Return type bool

instance_counter = 0

set_db_field_prefix(*prefix*, *override=False*)
Sets the graph property name prefix during document class construction.

set_property_name(*name*)
Sets the graph property name during document class construction.

This value will be ignored if db_field is set in __init__

Parameters **name** (*str*) – The name of this graph property

should_save(*first_save=False*)
Indicates whether or not the property should be saved based on it's save strategy.

Return type bool

to_database(*value*)

to_python(*value*)

validate(*value*)
Returns a cleaned and validated value. Raises a ValidationError if there's a problem

Return type Object

validator = <mogwai.properties.validators.RegexValidator object>

value_manager
alias of `BaseValueManager`

4.2.6 Relationships

class `mogwai.relationships.base.`**Relationship**(*edge_class*, *vertex_class*, *direction='BOTH'*, *strict=True*, *gremlin_path=None*, *vertex_callback=None*, *edge_callback=None*, *query_callback=None*, *create_callback=None*)
Define incoming and outgoing relationships that exist. Also enforce schema IN, OUT and BOTH directions

Warn if queries return schema violations.

allowed(*edge_type*, *vertex_type*)
Check whether or not the allowed Edge and Vertex type are compatible with the schema defined

Parameters

- **edge_type** – Edge Class
- **vertex_type** – Vertex Class

Type mogwai.models.Edge

Type mogwai.models.Vertex

Return type bool

create(**args*, ***kwargs*)
Creates a Relationship defined by the schema

Parameters

- **edge_params** (*dict*) – (Optional) Parameters passed to the instantiation method of the Edge
- **vertex_params** (*dict*) – (Optional) Parameters passed to the instantiation method
- **edge_type** (*mogwai.models.Vertex | None*) – (Optional) Edge class type, otherwise it defaults to the first Edge type known
- **edge_type** – (Optional) Vertex class type, otherwise it defaults to the first Vertex type known
- **callback** (*method*) – (Optional) Callback function to handle results

Return type tuple(mogwai.models.Edge, mogwai.models.Vertex) | Object

edges(**args*, ***kwargs*)
Query and return all Edges attached to the current Vertex

TODO: fix this, the instance method isn't properly setup :param limit: Limit the number of returned results :type limit: int | long :param offset: Query offset of the number of paginated results :type offset: int | long :param callback: (Optional) Callback function to handle results :type callback: method :rtype: List[mogwai.models.Edge] | Object

query(**args*, ***kwargs*)
Generic Query method for quick access

Parameters

- **edge_types** (*List[mogwai.models.Edge] | None*) – List of Edge classes to query against
- **callback** (*method*) – (Optional) Callback function to handle results

Return type mogwai.models.query.Query | Object

vertices(**args*, ***kwargs*)
Query and return all Vertices attached to the current Vertex

TODO: fix this, the instance method isn't properly setup :param limit: Limit the number of returned results :type limit: int | long :param offset: Query offset of the number of paginated results :type offset: int | long :param callback: (Optional) Callback function to handle results :type callback: method :rtype: List[mogwai.models.Vertex] | Object

mogwai.relationships.base.**requires_vertex**(*method*)

4.2.7 Save Strategies

class mogwai.properties.strategy.**SaveAlways**
Save this value every time the corresponding model is saved.

classmethod condition(*previous_value*, *value*, *has_changed=False*, *first_save=False*, *graph_property=None*)
Save this value every time the corresponding model is saved.

Return type bool

class mogwai.properties.strategy.**SaveOnChange**
Only save this value if it has changed.

classmethod condition(*previous_value*, *value*, *has_changed=False*, *first_save=False*, *graph_property=None*)
Always save this value if it has changed

Return type bool

class mogwai.properties.strategy.**SaveOnDecrease**
Save this value only if it is decreasing

classmethod condition(*previous_value*, *value*, *has_changed=False*, *first_save=False*, *graph_property=None*)
Only save this value if it is decreasing

Return type bool

class mogwai.properties.strategy.**SaveOnIncrease**
Save this value only if it is increasing

classmethod condition(*previous_value*, *value*, *has_changed=False*, *first_save=False*, *graph_property=None*)
Only save this value if it is increasing

Return type bool

class mogwai.properties.strategy.**SaveOnce**
Only save this value once. If it changes throw an exception.

classmethod condition(*previous_value*, *value*, *has_changed=False*, *first_save=False*, *graph_property=None*)
Always save this value if it has changed

Raises SaveStrategyException

Return type bool

class mogwai.properties.strategy.**Strategy**
Saving strategies for mogwai. These are used to indicate when a property should be saved after the initial vertex/edge creation.

classmethod condition(*previous_value*, *value*, *has_changed=False*, *first_save=False*, *graph_property=None*)
Default save strategy condition

Raises NotImplementedError

4.2.8 Property Validators

class mogwai.properties.validators.**BaseValidator**(*message=None*, *code=None*)

code = u'invalid'

message = u'Enter a valid value.'

class mogwai.properties.validators.**BooleanValidator**(*message=None*, *code=None*)

code = u'invalid'

message = u'Enter a valid Boolean.'

class mogwai.properties.validators.**DateTimeUTCValidator**(*message=None*, *code=None*)

code = u'invalid'

message = u'Not a valid UTC DateTime: {}'

class mogwai.properties.validators.**DateTimeValidator**(*message=None*, *code=None*)

code = u'invalid'

message = u'Not a valid DateTime: {}'

class mogwai.properties.validators.**DecimalValidator**(*message=None*, *code=None*)

code = u'invalid'

data_types = (<type 'float'>, <class 'decimal.Decimal'>)

message = u'Enter a valid number.'

class mogwai.properties.validators.**DictValidator**(*message=None*, *code=None*)

code = u'invalid'

message = u'Enter a valid dict'

class mogwai.properties.validators.**EmailValidator**(*regex=None*, *message=None*, *code=None*)

code = u'invalid'

message = u'Enter a valid email address: {}'

regex = <_sre.SRE_Pattern object at 0x2bac8f0>

class mogwai.properties.validators.**FloatValidator**(*message=None*, *code=None*)

code = u'invalid'

data_types = (<type 'float'>,)

message = u'Enter a valid number.'

class mogwai.properties.validators.**IntegerValidator**(*message=None*, *code=None*)

code = u'invalid'

data_types = (<type 'int'>, <type 'long'>)

message = u'Enter a valid number.'

class mogwai.properties.validators.**ListValidator**(*message=None*, *code=None*)

code = u'invalid'

data_types = (<type 'tuple'>, <type 'list'>)

message = u'Enter a valid list'

class mogwai.properties.validators.**LongValidator**(*message=None*, *code=None*)

code = u'invalid'

data_types = (<type 'int'>, <type 'long'>)

message = u'Enter a valid number.'

class mogwai.properties.validators.**NumericValidator**(*message=None*, *code=None*)

code = u'invalid'

data_types = (<type 'float'>, <type 'int'>, <type 'long'>, <class 'decimal.Decimal'>)

message = u'Enter a valid number.'

class mogwai.properties.validators.**PositiveIntegerValidator**(*message=None*, *code=None*)

code = u'invalid'

data_types = (<type 'int'>, <type 'long'>)

message = u'Enter a valid number.'

class mogwai.properties.validators.**RegexValidator**(*regex=None*, *message=None*, *code=None*)

code = u'invalid'

message = u'Enter a valid value.'

regex = u'‘

class mogwai.properties.validators.**StringValidator**(*message=None*, *code=None*)

code = u'invalid'

data_type = (<type 'basestring'>,)

message = u'Enter a valid string: {}'

class mogwai.properties.validators.**URLValidator**(*regex=None*, *message=None*, *code=None*)

code = u'invalid'

message = u'Enter a valid URL address: {}'

regex = <_sre.SRE_Pattern object at 0x2bab530>

4.2.9 Metrics

class mogwai.metrics.manager.**MetricManager**

Manages your Metric Agents

Properly inject mogwai metrics, while still allowing custom app metrics

count_calls(*fn*)

Decorator to track the number of times a function is called.

Parameters **fn** (*C{func}*) – the function to be decorated

Returns the decorated function

Return type C{func}

counters(*key=None*)
Yield Metric Reporter Counters

Parameters **key** (*basestring*) – The metric key to use

hist_calls(*fn*)
Decorator to check the distribution of return values of a function.

Parameters **fn** (*C{func}*) – the function to be decorated

Returns the decorated function

Return type C{func}

histograms(*key=None*)
Yield Metric Reporter Histograms

Parameters **key** (*basestring*) – The metric key to use

meter_calls(*fn*)
Decorator to the rate at which a function is called.

Parameters **fn** (*C{func}*) – the function to be decorated

Returns the decorated function

Return type C{func}

meters(*key=None*)
Yield Metric Reporter Meters

Parameters **key** (*basestring*) – The metric key to use

setup_reporters(*metric_reporters*)
Setup the Metric Reporter(s) for the MetricManager

start()
Start the Metric Reporter

stop()
Stop the Metric Reporter

time_calls(*fn*)
Decorator to time the execution of the function.

Parameters **fn** (*C{func}*) – the function to be decorated

Returns the decorated function

Return type C{func}

timers(*key=None*)
Yield Metric Reporter Timers

Parameters **key** (*basestring*) – The metric key to use

exception mogwai.metrics.manager.**MogwaiMetricsException**
Exception thrown when a metric system error occurs

args

message

class mogwai.metrics.base.**MetricsRegistry**(*clock=<built-in function time>*)
A single interface used to gather metrics on a service. It keeps track of all the relevant Counters, Meters, Histograms, and Timers. It does not have a reference back to its service. The service would create a L{MetricsRegistry} to manage all of its metrics tools.

clear()

counter(*key*)
Gets a counter based on a key, creates a new one if it does not exist.

@param key: name of the metric @type key: C{str}

@return: L{Counter}

dump_metrics()
Formats all of the metrics and returns them as a dict.

@return: C{list} of C{dict} of metrics

gauge(*key*, *gauge=None*)

get_metrics(*key*)
Gets all the metrics for a specified key.

@param key: name of the metric @type key: C{str}

@return: C{dict}

histogram(*key*)
Gets a histogram based on a key, creates a new one if it does not exist.

@param key: name of the metric @type key: C{str}

@return: L{Histogram}

meter(*key*)
Gets a meter based on a key, creates a new one if it does not exist.

@param key: name of the metric @type key: C{str}

@return: L{Meter}

timer(*key*)
Gets a timer based on a key, creates a new one if it does not exist.

@param key: name of the metric @type key: C{str}

@return: L{Timer}

class mogwai.metrics.base.**RegexRegistry**(*pattern=None*, *clock=<built-in function time>*)
A single interface used to gather metrics on a service. This class uses a regex to combine measures that match a pattern. For example, if you have a REST API, instead of defining a timer for each method, you can use a regex to capture all API calls and group them. A pattern like '^/api/(?P<model>)/d+/(?P<verb>)?$' will group and measure the following:

/api/users/1 -> users /api/users/1/edit -> users/edit /api/users/2/edit -> users/edit

clear()

counter(*key*)

dump_metrics()
Formats all of the metrics and returns them as a dict.

@return: C{list} of C{dict} of metrics

gauge(*key*, *gauge=None*)

get_metrics(*key*)
Gets all the metrics for a specified key.

@param key: name of the metric @type key: C{str}

@return: C{dict}

histogram(*key*)

meter(*key*)

timer(*key*)

exception mogwai.metrics.base.**MogwaiMetricsException**
Exception thrown when a metric system error occurs

args

message

class mogwai.metrics.base.**BaseMetricsReporter**(*registry=None*, *reportingInterval=300*, *metric_prefix=None*)
Base Metrics Reporter class

Customize this class for the specific metrics service, ie. Graphite, NewRelic, etc.

get_metrics(*timestamp=None*, **args*, ***kwargs*)
Default pyformance implementation to collect all the metrics from the registries.

Change this if you wan't customize for a particular metric collection system, ie. graphite, newrelic, etc.

Parameters **timestamp** (*long | int*) – use this timestamp instead of the generated timestamp when the method is run

Returns Timestamp and aggregated metrics from all registries.

Return type tuple(long | int, dict)

send_metrics(**args*, ***kwargs*)
Default pyformance implementation is to dump all metrics to STDOUT

Change this if you want to customize for a particular metric collection system, ie. graphite, newrelic, etc.

setup_registry(*registry=None*)
Setup the Metric Reporter with the given registry(s)

start()
Start the Metric Reporter

start_reporter(*reportingInterval*)
Start the Metric Reporter Agent

Kicks off a loop that every reportingInterval runs the *send_metrics* method.

Parameters **reportingInterval** (*float | long | int*) – The interval (number of seconds) on which to report the collected metrics.

stop()
Stop the Metric Reporter

mogwai.metrics.base.**ConsoleMetricReporter**
alias of *BaseMetricsReporter*

4.2.10 Compatibility

Utilizing the `six` library for Python 2 and 3 compatibility. More work and testing needs to be done to confirm Python 3 compatibility

4.2.11 Tools

class mogwai.tools.**BlueprintsWrapper**(*class_name=None*, *setup=None*, **args*, ***kwargs*)
Abstract implementation for using a Blueprints graph wrapper within a persisted transaction. Within this transaction *g* refers to the wrapped graph.

Parameters

- **class_name** (*str*) – the Blueprints Implementation class name
- **setup** (*str*) – an iterable with additional groovy statements that are executed upon initialization. (optional, default: [])

Returns the used RexPro connection pool (with default session)

Raises MogwaiBlueprintsWrapperException if no class name is provided

class mogwai.tools.**Factory**
Factory for Mogwai models.

ABSTRACT_FACTORY = True

exception mogwai.tools.**ImportStringError**(*import_name*, *exception*)
Provides information about a failed *import_string()* attempt.

exception = None

import_name = None

class mogwai.tools.**LazyImportClass**(*import_name*)
Lazily load and return a class

klass
Imports the class and caches

class mogwai.tools.**PartitionGraph**(*write=None*, *read=None*, **args*, ***kwargs*)
Wrapper for the Blueprints PartitionGraph, which is documented by https://github.com/tinkerpop/blueprints/wiki/Partition-Implementation

Parameters

- **write** (*str*) – the default read+write partition.
- **read** (*iterable*) – optional read partitions.

Returns the used RexPro connection pool (with default session)

Raises MogwaiBlueprintsWrapperException if no write partition is provided

class mogwai.tools.**SessionPoolManager**(*bindings=None*, *pool_size=10*)
A context manager that exposes a pool whose connections share the same session. If RexPro is used without concurrency, it is not necessary to use connections from the pool explicitly. *connection.execute_query* is patched in this case to use connections from the pool by default. In concurrent mode the pool always needs to be used explicitly, because we cannot rely on global context for patching.

Parameters

- **bindings** (*dict*) – variables that are available throughout the session (optional)
- **pool_size** (*int*) – the maximum number of simultaneous connections for this pool

Returns the used RexPro connection pool (with default session)

class mogwai.tools.**cached_property**(*func*, *name=None*, *doc=None*)
A decorator that converts a function into a lazy property. The function wrapped is called the first time to retrieve the result and then that calculated result is used the next time you access the value:

```
class Foo(object):

    @cached_property
    def foo(self):
        # calculate something important here
        return 42
```

The class has to have a *__dict__* in order for this property to work.

mogwai.tools.**import_string**(*import_name*, *silent=False*)
Imports an object based on a string. This is useful if you want to use import paths as endpoints or something similar. An import path can be specified either in dotted notation (`xml.sax.saxutils.escape`) or with a colon as object delimiter (`xml.sax.saxutils:escape`).

If *silent* is True the return value will be *None* if the import fails.

Parameters

- **import_name** – the dotted name for the object to import.
- **silent** – if set to *True* import errors are ignored and *None* is returned instead.

Returns imported object

mogwai.spec.**get_existing_indices**()
Find all Vertex and Edge types available in the database

mogwai.spec.**write_compiled_indices_to_file**(*filename*, *spec=None*)
Write the compile index specification to file

Parameters **filename** (*basestring*) – The file to write to

mogwai.spec.**write_diff_indices_to_file**(*filename*, *spec=None*)
Preview of index diff specification to write to file

Parameters **filename** (*basestring*) – The file to write to

mogwai.spec.**write_specs_to_file**(*filename*)
Generate and write a specification to file

Parameters **filename** (*basestring*) – The file to write to

4.2.12 Exceptions

mogwai.connection.**execute_query**(**args*, ***kwargs*)
Execute a raw Gremlin query with the given parameters passed in.

Parameters

- **query** (*str*) – The Gremlin query to be executed
- **params** (*dict*) – Parameters to the Gremlin query
- **connection** (*RexPro(Sync|Gevent|Eventlet)Connection or None*) – The RexPro connection to execute the query with
- **context** – String context data to include with the query for stats logging

Return type dict

mogwai.connection.**generate_spec**()
Generates a titan index and type specification document based on loaded Vertex and Edge models

mogwai.connection.**get_response**(*query*, *params*, *isolate*, *transaction*, *connection*, *connection_pool*)

mogwai.connection.**pop_execute_query_kwargs**(*keyword_arguments*)
pop the optional execute query arguments from arbitrary kwargs; return non-None query kwargs in a dict

mogwai.connection.**setup**(*host*, *graph_name=u'graph'*, *graph_obj_name=u'g'*, *username=u''*, *password=u''*, *metric_reporters=None*, *pool_size=10*, *concurrency=u'sync'*)
Sets up the connection, and instantiates the models

mogwai.connection.**sync_spec**(*filename*, *host*, *graph_name=u'graph'*, *graph_obj_name=u'g'*, *username=u''*, *password=u''*, *dry_run=False*)
Sync the given spec file to mogwai.

Parameters

- **filename** (*str*) – The filename of the spec file
- **host** (*str*) – The host the be synced
- **graph_name** (*str*) – The name of the graph to be synced
- **dry_run** (*boolean*) – Only prints generated Gremlin if True

Returns None

4.3 Examples

Common examples for using Mogwai

4.3.1 Quick Start

Usage

Note: This section provides a quick summary of mogwai features. A more detailed listing is available in the full documentation.

Setup Connection

You'll need to setup the connection to the graph database. Mogwai handles connection pooling for you, but you should handle load balancing using dedicated equipment.

```
from mogwai.connection import setup

# Without Authentication
setup('localhost')
# With authentication
#setup('localhost', username='rexster', password='rexster')
# With gevent support
#setup('localhost', concurrency='gevent')  # default is Standard Synchronous Python Sockets
# With eventlet support
#setup('localhost', concurrency='eventlet')  # default is Standard Synchronous Python Sockets
```

Defining Models

Models declare a Vertex or Edge model and attributes it should possibly contain.

```
from mogwai.models import Vertex, Edge
from mogwai.properties import String, Integer

_val = 0

def counter():
    global _val
    _val += 1
    return _val


class TestVertexModel(Vertex):
    element_type = 'test_vertex_model'

    name = String(default='test_vertex')
    test_val = Integer(default=counter)


class TestEdgeModel(Edge):
    label = 'test_edge_model'

    name = String(default='test_edge')
    test_val = Integer(default=counter)
```

Using Models

mogwai models can be utilized in a similar fashion to the way the Django ORM was constructed, but tailored for Graph Databases.

Creating Vertex or Edge

```
my_vertex_1 = TestVertexModel.create(name='my vertex')
my_vertex_2 = TestVertexModel.create(name='my other vertex')
my_edge = TestEdgeModel.create(outV=my_vertex_1, inV=my_vertex_2, name='my edge')
```

Retriveing a Vertex or Edge

```
# Get all the TestVertexModel Vertices
vertices = TestVertexModel.all()
# Get a subset of vertices by titan ID
vertices = TestVertexModel.all([1234, 5678, 9012])
# Get a vertex by titan ID
vertex = TestVertexModel.get(1234)

# Getting all Edges isn't currently supported
# Get a subset of edges by titan IDs
edges = TestEdgeModel.all(['123-UX4', '215-PX3', '95U-32Z'])
# get a single edge by titan ID
edge = TestEdgeModel.get('123-UX4')
```

```
# Get edge between two vertices
edge = TestEdgeModel.get_between(outV=my_vertex_1, inV=my_vertex_2)
```

Simple Traversals

Vertex Traversals

```
# Get All Edges from the vertex
edges = my_vertex_1.bothE()
# Get outgoing edges from the vertex
edges = my_vertex_1.outE()
# Get incoming edges to the vertex
edges = my_Vertex_1.inE()
# Specify an edge type for any edge traversal operation (works for outE, inE, bothE)
## By using models
test_model_edges = my_vertex_1.outE(TestEdgeModel)
## or by using manual labels
test_model_edges = my_vertex_1.outE('test_edge_model')

# Get all Vertices connected to the vertex
vertices = my_vertex_1.bothV()
# Get all vertices who are connected by edge coming into the current vertex (note uni-directed edges
vertices = my_vertex_1.outV()
# Get all vertices who are connected by edge coming from the current vertex (note uni-directed edges
vertices = my_vertex_1.inV()
# Specify an edge type for any edge traversal operation (works for outV, inV, bothV)
## By using models
test_model_vertices = my_vertex_1.outV(TestEdgeModel)
## or by using manual element types
test_model_vertices = my_vertex_1.outV('test_edge_model')
```

Edge Traversals

```
# Get the vertex which is from the outgoing side of the edge
vertex = my_edge.outV()
# Get the vertex which is from the incoming side of the edge
vertex = my_edge.inV()
```

4.3.2 Belongings Example

Here we illustrate creating models and utilizing the Relationship System in parallel with the Vertex Traversal system

```
from mogwai.connection import setup
from mogwai.models import Vertex, Edge
from mogwai import properties
from mogwai import relationships
from mogwai._compat import print_
import datetime
from pytz import utc
from functools import partial


setup('127.0.0.1')


class OwnsObject(Edge):
```

```
    label = 'owns_object'  # this is optional, will default to the class name

    since = properties.DateTime(required=True,
                                default=partial(datetime.datetime.now, tz=utc),
                                description='Owned object since')


class Trinket(Vertex):

    element_type = 'gadget'

    name = properties.String(required=True, max_length=1024)


class Person(Vertex):

    element_type = 'person'  # this is optional, will default to the class name

    name = properties.String(required=True, max_length=512)
    email = properties.Email(required=True)

    # Define a shortcut relationship method
    belongings = relationships.Relationship(OwnsObject, Trinket)


## Creation
# Create a trinket
trinket = Trinket.create(name='Clock')

# Create a Person
bob = Person.create(name='Bob Smith', email='bob@bob.net')

# Create the Ownership Relationship
relationship = OwnsObject.create(outV=bob, inV=trinket)


## Traversals
# Find out what bob owns
bob_owns_relationships = bob.belongings.vertices()
bob_owns_vertex_traversal = bob.outV(OwnsObject)

print_("With Relationships: Bob owns: {}".format(bob_owns_relationships))

# Another method for the same thing
print_("With Vertex Traversal: Bob owns: {}".format(bob_owns_vertex_traversal))
assert bob_owns_relationships == bob_owns_vertex_traversal


# Find out who owns the trinket
print_("With Vertex Traversal: Trinket is owned by: {}".format(trinket.inV(OwnsObject)))

# When was the trinket owned?
print_("Trinket has been owned since: {}".format(relationship.since))
```

4.3.3 Custom Gremlin Example

Here we illustrate creating models and utilizing the Gremlin system for custom methods

Python

```python
from mogwai.connection import setup
from mogwai.models import Vertex, Edge
from mogwai import properties
from mogwai import gremlin
from mogwai._compat import print_
import datetime
from pytz import utc
from functools import partial
import os

setup('127.0.0.1')


class IsFriendsWith(Edge):
    label = 'is_friends_with'

    since = properties.DateTime(required=True,
                                default=partial(datetime.datetime.now, tz=utc),
                                description='Owned object since')


class Person(Vertex):

    element_type = 'person'  # this is optional, will default to the class name
    gremlin_path = os.path.join(os.getcwd(), 'custom_gremlin.groovy')

    name = properties.String(required=True, max_length=512)
    email = properties.Email(required=True)

    friends_and_friends_of_friends = gremlin.GremlinMethod(method_name='friends_and_friends_of_friend
                                                           property=True,
                                                           defaults={'friend_edge_label': IsFriendsWi
    friends_and_friends_of_friends_table = gremlin.GremlinTable(method_name='friends_and_friends_of_f
                                                                property=True,
                                                                defaults={'friend_edge_label': IsFrie

    @property
    def friends(self):
        return self.bothV(IsFriendsWith)


## Creation
# Create the People
bob = Person.create(name='Bob', email='bob@bob.net')
alice = Person.create(name='Alice', email='alice@alice.net')
john = Person.create(name='John', email='john@john.net')
tim = Person.create(name='Tim', email='tim@tim.net')
kyle = Person.create(name='Kyle', email='kyle@kyle.net')

# Create Friendships
IsFriendsWith.create(outV=bob, inV=alice)
```

```
IsFriendsWith.create(outV=alice, inV=john)
IsFriendsWith.create(outV=alice, inV=tim)
IsFriendsWith.create(outV=tim, inV=kyle)

## Traverse
# All known direct friends with Bob
bobs_friends = bob.friends
print_("Bobs direct friends: {}".format(bobs_friends))
# Output:
# [ Person(element_type=person, id=8132, values={'name': Alice, 'email': alice@alice.net}) ]


# Friends and Friends of Friends using Custom Gremlin Method
bobs_friends_and_friends_of_friends = bob.friends_and_friends_of_friends
print_("Bobs Friends and Friends of Friends: {}".format(bobs_friends_and_friends_of_friends))

# Output:
# [
#   [ Person(element_type=person, id=8128, values={'name': Bob, 'email': bob@bob.net}),
#     Person(element_type=person, id=8132, values={'name': Alice, 'email': alice@alice.net}),
#     Person(element_type=person, id=8136, values={'name': John, 'email': john@john.net})
#    ], [
#     Person(element_type=person, id=8128, values={'name': Bob, 'email': bob@bob.net}),
#     Person(element_type=person, id=8132, values={'name': Alice, 'email': alice@alice.net}),
#     Person(element_type=person, id=8140, values={'name': Tim, 'email': tim@tim.net})
#    ]
# ]

for friend_set in bobs_friends_and_friends_of_friends:
    assert len(friend_set) <= 3
    assert friend_set[0] == bob
    assert friend_set[1] == alice
    assert kyle not in friend_set


# GremlinTable
print_("Using GremlinTable:")
friend_table = bob.friends_and_friends_of_friends_table
for row in friend_table:
    print_("{} is friends with {}, who is also friends with {}".format(row.person.name, row.friend.na

# Output:
# Using GremlinTable:
# Bob is friends with Alice, who is also friends with John
# Bob is friends with Alice, who is also friends with Tim
```

Gremlin

custom_gremlin.groovy

```
def friends_and_friends_of_friends(id, friend_edge_label) {
    /**
     * Traverse a Person and find any Friends of Friends
     *
     * :param id: Vertex id of the Person, if null abort
     * :param friend_edge_label: Edge label of the IsFriendsWith class
     * :returns: list[Person]
```

```
     */

     return g.v(id).out(friend_edge_label).loop(1){it.loops < 3}.path

}

def friends_and_friends_of_friends_table(id, friend_edge_label) {
    /**
     * Traverse a Person and find any Friends of Friends
     *
     * :param id: Vertex id of the Person, if null abort
     * :param friend_edge_label: Edge label of the IsFriendsWith class
     * :returns: list[Person]
     */
     return g.v(id).as('person').out(friend_edge_label).as('friend').out(friend_edge_label).simplePat
}
```

4.3.4 Serialization Example

Here we illustrate the serialization of the Vertex and Edge models (Pickle)

```
from mogwai.connection import setup
from mogwai.models import Vertex, Edge
from mogwai import properties
from mogwai import relationships
from mogwai._compat import print_
import datetime
from pytz import utc
from functools import partial
import pickle


setup('127.0.0.1')


class OwnsObject(Edge):

    label = 'owns_object'  # this is optional, will default to the class name

    since = properties.DateTime(required=True,
                                default=partial(datetime.datetime.now, tz=utc),
                                description='Owned object since')


class Trinket(Vertex):

    element_type = 'gadget'

    name = properties.String(required=True, max_length=1024)


class Person(Vertex):

    element_type = 'person'  # this is optional, will default to the class name

    name = properties.String(required=True, max_length=512)
    email = properties.Email(required=True)
```

```
    # Define a shortcut relationship method
    belongings = relationships.Relationship(OwnsObject, Trinket)


## Creation
# Create a trinket
trinket = Trinket.create(name='Clock')

# Create a Person
bob = Person.create(name='Bob Smith', email='bob@bob.net')

# Create the Ownership Relationship
relationship = OwnsObject.create(outV=bob, inV=trinket)


bob_serialized = pickle.dumps(bob)
print_("Bob Serialized: {}".format(bob_serialized))
deserialized_bob = pickle.loads(bob_serialized)
print_("Bob Deserialized: {}".format(deserialized_bob))
assert bob == bob_serialized

relationship_serialized = pickle.dumps(relationship)
print_("Relationship Serialized: {}".format(relationship_serialized))
deserialized_relationship = pickle.loads(relationship_serialized)
print_("Relationship Deserialized: {}".format(deserialized_relationship))
assert relationship == relationship_serialized


trinket_serialized = pickle.dumps(trinket)
print_("Trinket Serialized: {}".format(trinket_serialized))
deserialized_trinket = pickle.loads(trinket_serialized)
print_("Trinket Deserialized: {}".format(deserialized_trinket))
assert trinket == trinket_serialized
```

4.3.5 Unknown Properties Example

Here we illustrate the use of un-modeled properties on the Vertex models. Note, this also works for Edge models.

```
from mogwai.connection import setup
from mogwai.models import Vertex
from mogwai import properties

setup('127.0.0.1')


class Trinket(Vertex):

    element_type = 'gadget'

    name = properties.String(required=True, max_length=1024)


## Creation
# Create a trinket
# create with un-modeled property
trinket = Trinket.create(name='Clock', engraving='Bob Smith')
```

```
assert trinket['engraving'] == 'Bob Smith'

# get with un-modeled property
result = Trinket.get(trinket.id)
assert result['engraving'] == 'Bob Smith'

# change property from trinket (and persist to database)
trinket['engraving'] = 'Alice Smith'
assert trinket['engraving'] == 'Alice Smith'
trinket.save()
assert trinket['engraving'] == 'Alice Smith'

# delete property from trinke (and persist to database)
del trinket['engraving']
try:
    result = trinket['engraving']
    raise Exception("Property shouldn't exist")
except AttributeError:
    pass
trinket.save()
try:
    result = trinket['engraving']
    raise Exception("Property shouldn't exist")
except AttributeError:
    pass

# iterate through keys
keys = [key for key in trinket]
keys = [key for key in trinket.keys()]
values = [value for value in trinket.values()]
items = [item for item in trinket.items()]
```

4.3.6 Groovy Imports Example

Here we illustrate the use of importing in the groovy file, which applies these imports to all the function definitions.

```
import com.thinkaurelius.titan.core.util.*;

def test_method(a1, a2) {
    return a1, a2
}

def test_second_method(a1, a2) {
    return (a1..a2)
}
```

Here we illustrate the use of additional imports directly on specific GremlinMethod or GremlinValue:

```
from mogwai.connection import setup
from mogwai.models import Vertex
from mogwai import properties
from mogwai.gremlin import GremlinMethod

setup('127.0.0.1')


class Trinket(Vertex):
```

```
    element_type = 'gadget'

    name = properties.String(required=True, max_length=1024)

    test_method = GremlinMethod(path='example_groovy_imports.groovy', method_name='test_method',
                                imports=['com.thinkaurelius.titan.core.util.*'], classmethod=True)

# Call the test method:
result1, result2 = Trinket.test_method(1, 2)
assert result1 == 1
assert result2 == 2
```

4.3.7 Session Management Example

Session Wrapping

The default connection pool that is used to execute queries uses a new session for every request. This fully isolates queries from each other, which is a good idea in most cases. There are, however, also scenarios in which multiple queries need to share some kind of state. This can be done with a special kind of connection pool, which by default spawns connections that use the same session.

```
from mogwai import connection, properties
from mogwai.models import Vertex, Edge
from mogwai.tools import SessionPoolManager, BlueprintsWrapper, PartitionGraph
from mogwai.exceptions import MogwaiGremlinException

connection.setup('localhost')

##
#  Persist a session with SessionPoolManager
##

k = 10

with SessionPoolManager(bindings={'k': k}):
    gsk = connection.execute_query('"powers of ${k}"')
    pysk = "powers of {}".format(k)
    assert gsk == pysk

    kk = connection.execute_query("k * k")
    assert kk == k * k


##
#  Wrap the graph with a Blueprints Implementation
##

class BlueprintsWrapperVertex(Vertex):
    element_type = 'blueprints_wrapper_vertex'
    name = properties.String(required=True, max_length=128)


class BlueprintsWrapperEdge(Edge):
    element_type = 'blueprints_wrapper_edge'
    name = properties.String(required=True, max_length=128)
```

```
35
36 v0 = BlueprintsWrapperVertex.create(name="valid")
37
38 with BlueprintsWrapper(class_name='ReadOnlyGraph'):
39     v0 = BlueprintsWrapperVertex.get(v0._id)
40     try:
41         BlueprintsWrapperVertex.create(name="illegal")
42     except MogwaiGremlinException:
43         print "java.lang.UnsupportedOperationException: \
44                It is not possible to mutate a ReadOnlyGraph"
45
46
47 ##
48 #  Treat the graph as a Partition Graph
49 ##
50
51 with PartitionGraph(write='a'):
52     v1 = BlueprintsWrapperVertex.create(name="only in a")
53     v3 = BlueprintsWrapperVertex.create(name="started in a")
54
55 with PartitionGraph(write='b', read=['a']):
56     v2 = BlueprintsWrapperVertex.create(name="only in b")
57     e1 = BlueprintsWrapperEdge.create(outV=v2, inV=v1, name="only in b")
58     v3.name = "still in a"
59     v3.save()
60
61 with PartitionGraph(write='a'):
62     v1 = BlueprintsWrapperVertex.get(v1._id)
63     assert len(v1.bothE()) == 0
64     try:
65         BlueprintsWrapperVertex.get(v2._id)
66     except BlueprintsWrapperVertex.DoesNotExist:
67         print "vertex is located in partition B"
68
69 # outside of the session all the elements are accessible
70 print v1.bothE()[0]
71 # BlueprintsWrapperEdge(label=blueprints_wra..., values={'name': 'only in b'})
72
73 print dict(BlueprintsWrapperVertex.get(v3._id))
74 # {'_partition': 'a', 'name': 'still in a'}
```

Concurrent Session Wrapping

Note: The pool needs to be referenced explicitly by querying methods in the wrapper when the connection to Rexster is set up with e.g. *eventlet* or *gevent*.

The type of concurrency that is available in mogwai and rexpro is based on coroutines, which share the same global context. It would thus be unsafe to keep the session or its associated pool in a global variable. This would cause concurrently executed session wrappers to change the session key for all ongoing requests, also for those that should be executed within a different session.

All graph element methods that perform operations on the graph accept the optional keyword arguments *transaction*, *isolate*, and *pool*, which can override the defaults set by *execute_query*.

execute_query(query, params={}, transaction=True, isolate=True, pool=None

The following examples use *eventlet* for concurrent querying:

```python
from mogwai import connection, properties
from mogwai.models import Vertex, Edge
from mogwai.tools import SessionPoolManager, PartitionGraph
import eventlet

connection.setup('localhost', concurrency='eventlet')

##
#  Persist a session with SessionPoolManager
##

def isolation_test(scope):
    wrapper_config = {
        'bindings': {'scope': scope},
        'pool_size': 5
    }
    scope_values = []

    with SessionPoolManager(**wrapper_config) as pool:
        for i in range(7):
            scope_val = connection.execute_query(
                "scope *= 2",
                isolate=False,
                pool=pool
            )
            scope_values.append(scope_val)

    return scope, scope_values

pile = eventlet.GreenPile()
[pile.spawn(isolation_test, i) for i in range(10)]

for scope, scope_values in pile:
    assert scope_values == [scope * 2**i for i in range(1, 8)]


##
#  Treat the graph as a Partition Graph
##

class BlueprintsWrapperVertex(Vertex):
    element_type = 'blueprints_wrapper_vertex'
    name = properties.String(required=True, max_length=128)


class BlueprintsWrapperEdge(Edge):
    element_type = 'blueprints_wrapper_edge'
    name = properties.String(required=True, max_length=128)

n = 5

with PartitionGraph(write='a') as pool:
    av_pile = eventlet.GreenPile()
    for i in range(n):
        av_pile.spawn(BlueprintsWrapperVertex.create, name="only in a", pool=pool)
    vertices_in_a = list(av_pile)

with PartitionGraph(write='b', read=['a']) as pool:
```

```
    bv_pile = eventlet.GreenPile()
    for i in range(n):
        bv_pile.spawn(BlueprintsWrapperVertex.create, name="only in b", pool=pool)

    [pile.spawn(isolation_test, i) for i in range(n)]

    be_pile = eventlet.GreenPile()
    for vb in bv_pile:
        va = vertices_in_a.pop()
        scope, scope_values = pile.next()
        va[str(scope)] = scope_values
        av_pile.spawn(va.save, pool=pool)

        be_pile.spawn(BlueprintsWrapperEdge.create, outV=vb, inV=va,
                      name="only in b", pool=pool)

    vertices_in_a = [av for av in av_pile]
    edges_in_b = [be for be in be_pile]

for v in BlueprintsWrapperVertex.all():
    print v._id, dict(v), v.outE()

[pile.spawn(v.delete) for v in BlueprintsWrapperVertex.all()]
list(pile)
```

4.4 ChangeLog

Changes to the library are recorded here.

4.4.1 v0.7.5

- Loosen six version requirement, reflected updated dependency for loosened rexpro version

4.4.2 v0.7.4

- Fix minor bug with Edge.outV not returning Vertex - thanks to Leif Asgeirsson

4.4.3 v0.7.3

- Remove trace messages and clean-up developer logs

4.4.4 v0.7.1

- Fixed Rexpro Connection Pooling connection soft-closing on exception
- Fixed Relationship isolation bug (issue #10)

4.4.5 v0.7.0

- BluePrints PartitionGraph support - thanks to Alex Olieman (aolieman)

4.4.6 v0.6.8

- critical Relationships bugfix

4.4.7 v0.6.7

- fixed GremlinTable method to return a new Table object that composes Row(s)

4.4.8 v0.6.6

- find_by_value bugfix - thanks to Justin Hohner for identifying this bug.

4.4.9 v0.6.5

- Proper connection closing bugfix - thanks to Elizabeth Ramirez

4.4.10 v0.6.4

- Use blueprints element.setProperty method instead of deprecated addProperty - thanks to Kaisen Lin

4.4.11 v0.6.3

- as_dict() method now includes element id

4.4.12 v0.6.2

- Sane default __repr__ for Edges, feature parity to Vertex

4.4.13 v0.6.1

- Save Strategy bugfix - thanks to Nick Vollmar

4.4.14 v0.6.0

- Titan 0.5.0 support confirmed
- Tests updated to reflect changes, do note running tests against this requires titan 0.5.0

4.4.15 v0.5.0

- Python 3.3 and 3.4 compatibility - requires rexpro 0.3.0

4.4.16 v0.4.3

- Added support for groovy imports on file level, as well as through GremlinMethod/GremlinValue (*See Example*)

4.4.17 v0.4.2

- Added support for unknown properties and dictionary-like access (*See Example*)

4.4.18 v0.4.1

- Missing import fix
- setup.py typo fix

4.4.19 v0.4.0

- Support concurrent connections via gevent and eventlet through rexpro support in the 0.2.0 release
- All Properties support None when not required. Be aware when developing, especially around the Boolean property, since, when not required, it can actually be in 3 states (True, False and None) or (true, false and null in groovy)

4.4.20 v0.3.3

- Fixed and improved find_by_value for both Edge and Vertex

4.4.21 v0.3.2

- Fixed Circular Import
- Fixed DateTime UTC bug
- Wercker Integration
- Documentation updates

4.4.22 v0.3.1

- Bug Fixes
- Documentation updates

4.4.23 v0.3.0

- Utilize new rexpro.RexProConnectionPool
- Includes new rexpro green-thread friendly gevent.socket RexProSockets

4.4.24 v0.2.13

- setup.py `install_requires` hot fix

4.4.25 v0.2.12

- Public CI preview

4.4.26 v0.2.11

- Documentation Updates

4.4.27 v0.2.10

- Minor bug fixes

4.4.28 v0.2.9

Serializable models via pickle.

```
import pickle

vertex = MyTestVertex.create(name='test')
serialized_vertex = pickle.dumps(vertex)
deserialized_vertex = pickle.loads(serialized_vertex)
assert vertex == deserialized_vertex
```

4.4.29 v0.2.8

Re-Release of Mogwai to the public. Name change to Mogwai, which loosely means "gremlin". This is a major refactor of the original *thunderdome* library by Blake.

- Using RexPro, updated library to utilize RexPro and compatible with Titan 0.4.2
- Refactored library, changed the way properties are handled, validated and their associated save strategies.
- Removed vid and eid as primary keys, titan generates unique primary keys that we can utilize. Now accessible via Element._id or Element.id (the latter is a property shortcut to Element._id)
- Added groovy tests, updated gremlin base method for new _type_name
- Added interactive shell with some slight magic:

```
    Usage:
        python -m mogwai.shell --host localhost
    For more help see:
        python -m mogwai.shell --help
    Also HELP is available in the shell
```

- Preview of index specification system, initial commit
- Relationship system, includes generic query method, create relationship method and strict relationship checker
- Fixed groovy files to only use local variables in core structure, will prevent Concurrent Global variable scope locks
- Special Enum Vertex metaclass now available. ie. *MyVertex.MY_ENUM* will be able to resolve to an actual database entry
- Performance monitoring tools now available - Customizable for different metric reporting mechanisms, ie, console, logs, graphite, newrelic.
- Apache 2.0 License

4.5 Contributing

4.5.1 License

mogwai is distributed under the Apache 2.0 License.

4.5.2 Issues

Issues should be opened through Bitbucket Issues; whenever possible, a pull request should be included.

4.5.3 Pull Requests

General Rules:

- All Tests must pass
- Coverage shouldn't decrease
- All Pull Requests should be rebased against master **before** submitting the PR.

All pull requests should pass the test suite, which can be launched simply with:

```
$ make test
```

Note: Running test requires *nosetests*, *coverage*, *six*, *pyformance*, *rexpro* and *factory_boy*. As well an available titan server.

In order to test coverage, please use:

```
$ pip install coverage
$ coverage erase; make coverage
```

4.5.4 Test Server

mogwai was designed for Titan, other graph databases that utilize Blueprints may be compatible, but further testing would be needed.

Currently Titan 0.4.4 is known to work and can be downloaded: Titan.

4.6 Ideas

We'll keep a running list of ideas for future enhancements. Please note this is likely incomplete and not always up-to-date with the issue tracker. Also these are in no-particular order.

- **Titan 0.5.x + support –> TinkerPop3 and Gremlin3 Support**
  - Websocket support for streaming results (generators)
  - Build in pluggable gevent/eventlet support for streaming results – mogwai operations should be asynchronous
- **Improved Index/Constraint Specification System**
  - Create Faunus or runnable job against the database to enforce a specification

- **Support Migrations - The 'South' project lends inspiration here.**
- **Add Relationship attribute magic**
  - Schema enforcement for Faunus
  - Creation/Traversal enforcement
- **Integrate additional common metric collectors**
- **Filter lazy-evaluated gremlin query.**
  - Perhaps: MyVertexModel.query().filter(someval__lte=2)
- **Database Export/Import functionality**
- **Enum MetaClass optional memcache/etc support.**
- **Gremlin Query builder** * Optional groovy script generator based on query.
- **Groovy script inspections tool for optimization hints, common algorithms, possible syntax errors.**

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

m

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W