RethinkORM Documentation

Release 0.1.0

Joshua P Ashby

August 08, 2013

CONTENTS

Build status: RethinkORM is a small wrapper class to help make working with documents in RethinkDB easier, and in a more Pythonic way.

I recently found RethinkDB and was amazed at how easy everything seemed to be, however one thing that I've missed is how the data is just a Python `List` or `Dict` rather than a full wrapper class. So I figured a good way to learn the general use of the Python RethinkDB driver was to write a general wrapper class that functioned a bit like an ORM, providing some easier to work with data and objects.

Unittests are included, and the code should be PEP8 compliant. The tests are automatically ran each commit, thanks to travis-ci.org and this documentation is kindly hosted and automatically rebuilt by readthedocs.org.

Gittip if you like the work I do and would consider a small donation to help fund me and this project:

CHAPTER

ONE

A FEW MINOR WARNINGS

1. I'm only a second year university student, and software isn't even my major; I'm working towards an Electrical and Computer Engineering degree, so not only do I have limited time to keep this maintained, but I also probably won't write the best code ever.
2. This takes some influence from the Python Django RethinkDB ORM and other ORM systems, however I haven't really followed a standard pattern for the interface for this module. If someone wants to make this more standardized feel free to, and just submit a pull request, I'll look it over and probably will give it the go ahead. For more information see below.
3. This is a very early release, things might break, and the code is honestly a little childish at best. In other words: It'll hopefully get better, but it might be a little limited right now.

CHAPTER

TWO

QUICK START:

First we need to make an object which will represent all of our data in a specific table, along with getting a connection to RethinkDB started.

```
import rethinkdb as r
from rethinkORM import RethinkModel

r.connect(db="props").repl()


class tvProps(RethinkModel):
    table = "stargate_props"
```

For more information on what class properties are available to change, see *rethinkORM*

2.1 Inserting/creating an entry

```
dhdProp = tvProps(what="DHD", planet="P3X-439", description="Dial HomeDevice")
dhdProp.id="DHD_P3X_439"
dhdProp.save()
```

2.2 Updating an entry

```
updatedProp = tvProps("DHD_P3X_439")
updatedProp.description="""Dial Home Device from the planel P3X-439, where an
    Ancient Repository of Knowledge was found, and interfaced with by Colonel
    Jack."""
updatedProp.save()
```

2.3 Deleting an entry

```
oldProp = tvProps("DHD_P3X_439")
oldProp.delete()
```

CHAPTER

THREE

CONTRIBUTING

Submit a pull request or open an issue. Most things I probably won't have time to get around to looking at too deeply, so if you want it fixed, a pull request is the way to go. Besides that, I'm releasing this under the GPLv3 License as found in the `LICENSE.txt` file. Enjoy!

CHAPTER

FOUR

DOC CONTENTS

4.1 rethinkORM

4.1.1 `rethinkModel` Module

class `rethinkORM.rethinkModel.`**`RethinkModel`**(*id=False*, ***kwargs*)

Emulates a python object for the data which is returned from rethinkdb and the official Python client driver. Raw data from the database is stored in _data to keep the objects namespace clean. For more information look at how _get() and _set() function in order to keep the namespace cleaner but still provide easy access to data.

This object has a __repr__ method which can be used with print or logging statements. It will give the id and a representation of the internal _data dict for debugging purposes.

`table` = ''

The table which this document object will be stored in

`primaryKey` = 'id'

The current primary key of the table

`durability` = 'soft'

Can either be Hard or Soft, and is passed to RethinkDB

`non_atomic` = False

Determins if the transaction can be non atomic or not

`upsert` = True

Will either update, or create a new object if true and a primary key is given.

`__init__`(*id=False*, ***kwargs*)

Initializes the main object, if *id* is in kwargs, then we assume this is already in the database, and will try to pull its data, if not, then we assume this is a new entry that will be inserted.

(Optional, only if not using .repl()) *conn* or *connection* can also be passed, which will be used in all the .run() clauses.

`finishInit`()

A hook called at the end of the main *__init__* to allow for custom inherited classes to customize their init process without having to redo all of the existing int. This should accept nothing besides *self* and nothing should be returned.

`__delitem__`(*item*)

Deletes the given item from the objects _data dict, or if from the objects namespace, if it does not exist in _data.

__contains__(*item*)
Allows for the use of syntax similar to:

```
if "blah" in model:
```

This only works with the internal _data, and does not include other properties in the objects namepsace.

classmethod new(***kwargs*)
Creates a new instance, filling out the models data with the keyword arguments passed, so long as those keywords are not in the protected items array.

classmethod create(*id=None*, ***kwargs*)
Similar to new() however this calls save() on the object before returning it.

classmethod find(*id*)
Loads an existing entry if one can be found, otherwise an exception is raised.

Parameters **id** (*Str*) – The id of the given entry

Returns *cls* instance of the given *id* entry

save()
If an id exists in the database, we assume we'll update it, and if not then we'll insert it. This could be a problem with creating your own id's on new objects, however luckily, we keep track of if this is a new object through a private _new variable, and use that to determine if we insert or update.

delete()
Deletes the current instance. This assumes that we know what we're doing, and have a primary key in our data already. If this is a new instance, then we'll let the user know with an Exception

__repr__()
Allows for the representation of the object, for debugging purposes

protectedItems
Provides a cleaner interface to dynamically add items to the models list of protected functions to not store in the database

4.1.2 Subpackages

rethinkORM.tests

To get started and make sure this all works, please make sure you have Python nose installed.

```
nosetests rethinkORM -v -s
```

This will run the all the tests, not capturing `stdout` and being verbose, in case anything goes wrong, or if you modify the tests. Please note, tests are subject to a lot of changes, and this may not always be the same command.

If you want to also check the PEP8 validity of the code, you can run:

```
pep8 rethinkORM
```

or, if you have tissue installed you can run a PEP8 check with the rest of the test suite like so:

```
nosetests rethinkORM -v -s --with-tissue
```

How the tests work (or should, if more are written):

There is a setup fixture that creates a database called `model` and within that creates a table `stargate`. Then each test works on entries which get stored in this database and table. When everything is done, the teardown fixture is ran

to clean up and delete the whole database `model`. Each test should be broken down into basic actions, for example there are currently tests for:

- inserting a new entry
- modifying that entry
- deleting that entry
- inserting an entry where the primary key is `None` or a null value.

`test_model` Module

Test suite for the model

`rethinkORM.tests.test_model.`**`setup`**`()`
Sets up the connection to RethinkDB which we'll use in the rest of the tests, and puts it into .repl() mode so we don't have to pass the model object a connection. After that, we create a new database called *model* and within that a table called *stargate* and sets the database to use *model*.

`rethinkORM.tests.test_model.`**`teardown`**`()`
Drops the whole *model* database, since it's no longer needed now that the tests are done.

class `rethinkORM.tests.test_model.`**`gateModel`**`(`*id=False*, ***kwargs*`)`
Bases: `rethinkORM.rethinkModel.RethinkModel`

Sample document object which represents the documents within the table *stargate*.

class `rethinkORM.tests.test_model.`**`base`**
Bases: `object`

Base test object to help automate some of the repetitive work of reloading a document to ensure the model matches the test data. Also takes care of deleting the document if *cleanupAfter* is *True*

`cleanupAfter` **= False**
Should the document created by this test be deleted when over?

`loadCheck` **= True**
Should the document be reloaded and have all it's data checked against?

`whatToLoad` **= []**
If loadCheck is true, fill this out with strings of the data keys to check the model against.

`model` **= None**
The model being used for this test

`data` **= None**
The data being used for this test. Please at least include an ID

`action``()`
Override this with your own function to do whatever you want for the test

`load``()`
Override this to do a custom load check. This should find the key you created or modified in *action()* and check it's values to ensure everything was set correctly. By default this loads the model with the test objects *data["id"]* and uses *whatToLoad* to run checks against the data and the model.

`cleanup``()`
Override this to set a custom cleanup process. By default this takes the key that was generated in *action()* and calls the models *.delete()* function.

class rethinkORM.tests.test_model.**insert_test**
Bases: rethinkORM.tests.test_model.base

Tests the basic ability to make a new model instance, and save it to the Database

model
alias of gateModel

action()
Creates a new object, and inserts it, using *.save()*

class rethinkORM.tests.test_model.**modify_test**
Bases: rethinkORM.tests.test_model.base

Tests the ability to load, modify and save a model correctly

model
alias of gateModel

action()
Next, we get the object again, and this time, we modify it, and save it.

rethinkORM.tests.test_model.**insertBadId_test**(**arg*, ***kw*)
Here we test to make sure that if we give a primary key of type *None* that we are raising an exception, if we don't get an exception then something is wrong since the primary key shouldn't be allowed to be *None*

rethinkORM.tests.test_model.**insertIdAndData_test**(**arg*, ***kw*)
Make sure that the model raises an Exception when a key and data are provided

class rethinkORM.tests.test_model.**new_classmethod_test**
Bases: rethinkORM.tests.test_model.base

Tests the new() classmethod of the model

model
alias of gateModel

class rethinkORM.tests.test_model.**create_classmethod_test**
Bases: rethinkORM.tests.test_model.base

Tests the create() classmethod of the model

Same as the new() classmethod test however we don't have to explicitly tell the model to save

model
alias of gateModel

class rethinkORM.tests.test_model.**find_classmethod_test**
Bases: rethinkORM.tests.test_model.base

Tests the find() classmethod of the model

model
alias of gateModel

4.2 Indices and tables

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

PYTHON MODULE INDEX

r