RethinkORM

Release 1.0.0

April 07, 2014

Contents

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

When I found RethinkDB, I was amazed at how easy everything seemed to be, however one thing that I missed was how the data was 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.

Changes from Version 0.2.0

1. Relicensed from GPL v3 to MIT.
2. Several names have changed, primarially: *protectedItems* is now *protected_items*, *primaryKey* is now *primary_key*, *orderBy* in collections is now *order_by*. There are probably others that I'm missing however.
3. The ability to join tables or models within a collection have been removed for now (it was mostly broken anyways).
4. The find classmethod on models has been removed (For now, until I come up with a better way of doing this).
5. *fromRawEntry* is not outdated, and can be replaced by just instantiating a new model with the data.
6. The models no longer keep track of if a document is new and just use the RethinkDB drivers *upsert* ability to insert or update a document.
7. Passing a key and data will now no longer raise an exception, but instead return a new model.
8. Providing only *id* as a keyword argument to the model will cause it to assume the document is in the database, and it will attempt to get that document.

CHAPTER 2

Installation:

```
pip install RethinkORM
```

CHAPTER 3

Quick Start

This library provides the `RethinkModel` class which provides a base class that works as a mapper between a rethink document and a Python object, along with the `RethinkCollection` class to represent a group of documents which match a ReQL query.

First, lets setup the basic rethink connection and get a sample database and table made. I choose to go with this pattern, providing an option for passing the connection object into the models, because I don't quite like the ORMs which try to fully replace the underlying driver. This also gives a nice easy way to use standard ReQL for operations that are beyond the scope of what these models are for.

```
>>> import rethinkdb as r
>>> from rethinkORM import RethinkModel
>>> from rethinkORM import RethinkCollection
>>> conn = r.connect('localhost', 28015)
>>> a = r.db_create("example").run(conn)
>>> conn.use("example")
>>> conn.repl()
<rethinkdb.net.Connection ...>

>>> a = r.table_create("stargates").run()
```

Now we can create a model that we'll be working with.

```
>>> class Stargate(RethinkModel):
...   table = "stargates"
```

And now we can create an actual document using this model. Create is a classmethod that will return a new *Stargate* object with the ID of the newly created document, with the passed data. In this case: *location*, and *description*.

```
>>> earth_gate = Stargate.create(location="Earth", description="The first of two gates found on eart
```

If you do not call *.repl()* on the rethink connection, you can also pass conn to the model like so:

```
>>> earth_gate = Stargate.create(location="Earth", description="The first of two gates found on eart
```

If you did not want to immediately save the document upon creation you can instead call the *.new()* classmethod:

```
>>> atlantis_gate = Stargate.new(location="City of Atlantis", description="The gate in the gate room
```

You can then make changes to this document, and when you are finished, call *.save()* which will save the document and set the objects ID to the newly created documents ID.

```
>>> atlantis_gate.save()
True
```

Getting a collection of documents is also easy through the use of `RethinkCollection`. Simply give `RethinkCollection` the classref of your model, in our case *Stargate* and look at the documents property. This will fetch all of the documents within the table defined by the *Stargate* model (the *stargates* table in this case). `RethinkCollection` also provides several helpers such as accepting a pre built ReQL query, limiting of results, sorting the documents and filtering them based off of field values.

```
>>> gates = RethinkCollection(Stargate)
>>> gates.documents
[<RethinkModel ...>, <RethinkModel ...>]

>>> a = r.db_drop("example").run()
```

CHAPTER 4

A Few Minor Warnings

1. I am a second year university student working towards a 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. While this is now on its second major version, the code is still maturing a bit so chances are stuff will still break. Again, see below for information about contributing patches or features.

CHAPTER 5

Doc Contents

5.1 Models

The model is the core of everything RethinkORM deals with. All data returned from RethinkDB is eventually wrapped in the model before being returned to the end user. It provides an pythonic, object style interface for the data, exposing methods to save and update documents along with creating new ones.

5.1.1 General Usage

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"
```

Inserting/creating an entry

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

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()
```

Deleting an entry

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

5.1.2 `RethinkORMException`

class `rethinkORM.`**`RethinkORMException`**

Gotta catch them all!

5.1.3 `RethinkModel`

class `rethinkORM.`**`RethinkModel`**(*id=None*, ***kwargs*)

This mostly emulates a basic python *object* for the returned data from RethinkDB. You can use both attribute access and item access to interact with the data.

`RethinkModel.`**`table`** **= ‘’**

The table which this document object will be stored in

`RethinkModel.`**`primary_key`** **= ‘id’**

The current primary key in use on the table. This defaults to *id* as RethinkDB will default to using *id* as the primary key.

> **Warning:** In pre v1.0.0 releases, this was called *primaryKey*

`RethinkModel.`**`durability`** **= ‘soft’**

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

`RethinkModel.`**`__init__`**(*id=None*, ***kwargs*)

Initializes the main object, if *id* is the only thing passed then we assume this document is already in the database and grab its data, otherwise we treat it as a new object.

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

`RethinkModel.`**`finish_init`**()

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.

`RethinkModel.`**`__delitem__`**(*item*)

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

`RethinkModel.`**`__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 `RethinkModel.`**`new`**(*id=None*, ***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 `RethinkModel.`**`create`**`(`*id=None*, ***kwargs*`)`
Similar to new() however this calls save() on the object before returning it, to ensure that it is already in the database. Good for make and forget style calls.

`RethinkModel.`**`save`**`()`
Update or insert the document into the database.

`RethinkModel.`**`delete`**`()`
Deletes the current instance, if its in the database (or try).

`RethinkModel.`**`__repr__`**`()`
Allows for the representation of the object, for debugging purposes

`RethinkModel.`**`protected_items`**
Provides a cleaner interface to dynamically add items to the models list of protected functions to not store in the database.

> **Warning:** In the pre v1.0.0 releases, this was called *protectedItems*

5.2 Collections

Collections provide a quick and easy way to interact with many documents of the same type all at once. They also provide a mechanism for basic joins across one addition table (due to current limitations of RethinkDB and how it handles joins). Collections act like python *lists* of `RethinkModel` objects, but provide an easy interface to order, filter, and limit the results.

5.2.1 General Usage

Collections are pretty simplistic objects. Just instantiate a new object with the model which you are constructing the collection from.

```
collection = RethinkCollection(gateModel)
```

Note: Optionally you can also pass a dictionary which will be used as a filter. For more information on how filters work, please see the RethinkDB docs

Order the Results

```
collection.orderBy("episodes", "ASC")
```

Limit the Results

```
collection.limit(10, 5) # Limit to 10 results, starting after the 5th one.
```

Fetch the Results

```
result = collection.fetch()
```

result acts like a bit like a normal python *list*, containing all of the documents which are part of the collection, all pre-wrapped in a `RethinkModel` object.

5.2.2 `RethinkCollection`

class `rethinkORM.`**`RethinkCollection`**(*model*, *filter=None*, *query=None*, *conn=None*)

A way to fetch groupings of documents that meet a criteria and have them in an iterable storage object, with each document represented by *RethinkModel* objects

`RethinkCollection.`**`__init__`**(*model*, *filter=None*, *query=None*, *conn=None*)

Instantiates a new collection, using the given models table, and wrapping all documents with the given model.

Filter can be a dictionary of keys to filter by, a lambda or a similar statement, see: RethinkDB Filters for more information

A pre built query can also be passed in, to allow for better control of what documents get included in the final collection.

Parameters

- **model** – A RethinkModel object to be used to wrap all documents in
- **filter** – If provided, it will be passed using the ReQL .filter command
- **query** – An optional pre built ReQL query to be used

`RethinkCollection.`**`order_by`**(*key*, *direction='desc'*)

Allows for the results to be ordered by a specific field. If given, direction can be set with passing an additional argument in the form of "asc" or "desc"

Warning: In pre v1.0.0 releases, this was named *orderBy*

Parameters

- **key** – The key to sort by
- **direction** – The direction, DESC or ASC to sort by

`RethinkCollection.`**`limit`**(*limit*, *offset=0*)

Limits the results to the given offset (0 if not given) and the stated limit.

Parameters

- **limit** – The number of documents that the results should be limited to
- **offset** – The number of documents to skip

`RethinkCollection.`**`fetch`**()

Fetches the query results and wraps the documents in the collections model.

Documents can then be accessed through the standard *collection[index]* or with a *for* loop:

```
for doc in collection:
    pass
```

CHAPTER 6

Contributing

All code for this can be found online at github. If something is broken, or a feature is missing, please 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. In your pull request please provide an explaniation as to what your request is for, and what benefit it provides. Also please try to follow PEP8 as best a possible.

Unittests are included in the *tests/* directory and are ran every commit thanks to travis-ci.org and this documentation is kindly hosted and automatically rebuilt by readthedocs.org. Test coverage is also provided by coveralls.io.

If you would like to run the tests on your own all you have to do is:

```
nosetests
```

Note: Tests require nose and coverage to be installed.

License

This is released this under the MIT License as found in the `LICENSE.txt` file. Enjoy!

Warning: Heads up, version 1.0.0 has a different license than previous releases: The pre v1.0.0 releases were licensed under the GPL v3 License.

7.1 Versioning

This project will try to follow the semantic versioning guide lines, as laid out here: SemVer, as best as possible.

7.2 Thanks

Shout outs to these people for contributing to the project:

1. scragg0x
2. grieve

7.3 Indices and tables

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