# Repocket Documentation

*Release 0.0.1*

Gabriel Falcão

Apr 29, 2018

# Contents

Repocket is an active-record for python that is completely inspired in the API of CQLEngine, which is seemingly inspired in the Django ORM API.

Repocket is also a very small library and you will master it in a second.

CHAPTER 1

# Introduction

Repocket is an active record that let's you use redis as main data store.

Redis is commonly seen as a ephemeral, cache-purposed in-memory database.

But the reality is that redis is a data structure server.

In the same way that python has the `int`, `float`, `unicode`, `list`, `set` and `dict` builtin types, redis has equivalent datastructures, and some really cool functions to manipulate them in an optimized way.

Repocket lets you declare your models in a Django-fashioned way, automatically validate your fields, store in redis and retrieve them in a very elegant way.

Also Repocket is ready for application needs like "foreign key".

Nobody likes foreign keys, relational databases get slow and complex because of relationships and constraints. In fact, the reason that all the logic of validation, contraints and consistency checks was built in SQL databases is that back in the day we didn't have great application frameworks and thousands of open source tools to help us write great, reliable software.

But that changed, now you can use database servers just to store your data, and all the consistency checks and validations can live in your application code.

Repocket supports *"pointers"* which are references from one active record to another, also it will automatically retrieve the directly-related objects for you when you retrieve data from redis.

Here is a full example with all the supported field types of repocket:

```
from repocket import attributes
from repocket import ActiveRecord

 class Project(ActiveRecord):
     name = attributes.Unicode()
     git_uri = attributes.Bytes()
     metadata = attributes.JSON()

 class Build(ActiveRecord):
     id = attributes.AutoUUID()
```

(continues on next page)

(continued from previous page)

```
    project = attributes.Pointer(Project)
    started_at = attributes.DateTime()
    ended_at = attributes.DateTime()
    stdout = attributes.ByteStream()
    stderr = attributes.ByteStream()
```

# CHAPTER 2

# Tutorial

Here you will learn how to become fluent inrepocket in just a couple of minutes.

## 2.1 Installing

```
pip install repocket
```

## 2.2 Configuring the connection

Repocket uses a global connection pool where all the connections will be shared when possible.

In your application code you will have to configure how repocket connects, but you will do it only once, to show you how, imagine that this is your own application code:

```
>>> from repocket.connections import configure
>>> configure.connection_pool(hostname='myredis.cooldomain.com', port=6379, db=0)

# at this point you're ready to use your declared models
```

## 2.3 Declaring Models

Repocket provides a model interface that looks just like Django, but the field types are super simplified.

Here is how you declare a model:

```
>>> from repocket import attributes
>>> from repocket import ActiveRecord
>>>
```

(continues on next page)

(continued from previous page)

```
>>>
>>> class User(ActiveRecord):
...     name = attributes.Unicode()
...     house_name = attributes.Unicode()
...     email = attributes.Bytes()
...     password = attributes.Bytes()
...     metadata = attributes.JSON()
```

If you were in Django you would then need to run `syncdb` to have a SQL table called `User` with the declared fields. But *this ain't Django, ight?*

At this point you are ready to start saving user data in redis.

By default the attributes of the your model are actively saved in a `hash` redis datastructure.

Repocket *currenty* also supports another attribute called `ByteStream` that will seamlessly store the value in a string, so that you can `APPEND` more bytes to it with a single call.

But we will get there soon enough, for now let's understand how to save a new user and how it will be saved inside of redis.

## 2.4 Persisting Data

Let's save a `User` instance in redis:

```
>>> import bcrypt

>>> harry = User.create(
...     id='970773fa-4de1-11e5-86f4-6c4008a70392',
...     name='Harry Potter',
...     email='harry@hogwards.uk',
...     house_name='Gryffindor',
...     password=bcrypt.hashpw(b'hermione42', bcrypt.gensalt(10)),
...     metadata={
...         'known_tricks': [
...             "Protego",
...             "Expelliarmus",
...             "Wingardium Leviosa",
...             "Expecto Patronum"
...         ]
...     }
... )
>>> ron = User.create(
...     id='40997aa4-71fc-4ad3-b0d7-04c0fac6d6d8',
...     name='Ron Weasley',
...     house_name='Gryffindor',
...     email='ron@hogwards.uk',
...     password=bcrypt.hashpw(b'hermione42', bcrypt.gensalt(10)),
...     metadata={
...         'known_tricks': [
...             "Protego",
...             "Expelliarmus",
...         ]
...     }
... )
```

## 2.5 Retrieving an item by its id

```
>>> harry = User.objects.get(id='970773fa-4de1-11e5-86f4-6c4008a70392')
>>> harry.metadata
{
    'known_tricks': [
        "Protego",
        "Expelliarmus",
        "Wingardium Leviosa",
        "Expecto Patronum"
    ]
}
```

## 2.6 Manipulating in-memory data

You can get the valus of an instance with either `.attribute`` notation or `["attribute"]`.

```
>>> harry = User.objects.get(id='970773fa-4de1-11e5-86f4-6c4008a70392')
>>> harry.id
UUID('970773fa-4de1-11e5-86f4-6c4008a70392')
```

```
>>> harry['id']
UUID('970773fa-4de1-11e5-86f4-6c4008a70392')
```

## 2.7 Deleting a record from redis

The `delete()` method returns an integer corresponding to the number of redis keys that were deleted as result.

```
>>> harry = User.objects.get(id='970773fa-4de1-11e5-86f4-6c4008a70392')
>>> harry.delete()
1
```

## 2.8 Retrive multiple items with filter

```
>>> results = User.objects.filter(house_name='Griffindor')
>>> len(results)
2
>>> results[0].name
'Harry Potter'
>>> results[1].name
'Ron Weasley'
```

**Note:** The order in which the elements are returned by `filter()` cannot be guaranteed because the id is a *uuid*. Use the `.order_by()` method

The `filter()` method returns a `ResultSet` object, which is a list with superpowers. The main superpower is the ability to order the results.

```
>>> results = User.objects.filter(house_name='Griffindor').order_by('-name')
>>> len(results)
2
>>> results[0].name
'Ron Weasley'
>>> results[1].name
'Harry Potter'
```

CHAPTER 3

# Serialization Rules

Repocket stores your data consistently with its original field type. Under the hood repocket stores everything as json, in a way or another.

Here you will the rules followed by repocket so that your data content is pristine.

## 3.1 How it gets stored in redis

Later in this documentation you will learn the rules that repocket follows to generate redis keys, for now know that the `.save()` method returns a dictionary containing all the redis keys used to store that one model instance's data.

Because we don't have any `ByteStream` fields in the `User` model definition, all the data will be declared in a single *hash* in redis. So lets check what its redis key looks like:

```
>>> harrys_keys
{
    "hash": "repocket:tests.functional.test_active_record:User:970773fa-4de1-11e5-
↪86f4-6c4008a70392",
    "strings": {}
}
```

## 3.2 The guts of the data

Now you know that the redis key for the *hash* is `repocket:tests.functional.test_active_record:User:970773fa-4de1-11e5-86f4-6c4008a70392`, so now you can check what is in redis:

```
$ redis-cli --raw HGETALL repocket:tests.functional.test_active_record:User:970773fa-
↪4de1-11e5-86f4-6c4008a70392
 email
 {"type": "Bytes", "value": "harry@hogwards.uk", "module": "repocket.attributes"}
```

(continues on next page)

(continued from previous page)

```
name
{"type": "Unicode", "value": "Harry Potter", "module": "repocket.attributes"}
password
{"type": "Bytes", "value": "somethingsecret", "module": "repocket.attributes"}
id
{"type": "AutoUUID", "value": "970773fa-4de1-11e5-86f4-6c4008a70392", "module":
↪"repocket.attributes"}
metadata
{"type": "JSON", "value": "{'known_tricks': ['Protego', 'Expelliarmus', 'Wingardium␣
↪Leviosa', 'Expecto Patronum']}", "module": "repocket.attributes"}
```

Awesome! You can see your data in redis, you can notice how repocket stores the data in a json object with metadata that describes the stored type. You can learn more in the *Serialization Rules* chapter

**Note:** the metadata field is an `attributes.JSON()` field, so it can store any builtin python type, and automatically serializes it. It's a great example of how flexible you can be with repocket.

# CHAPTER 4

# API Reference

## 4.1 Attributes

**class** `repocket.attributes.`**`Attribute`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Repocket treats its models and attributes as fully serializable. Every attribute contains a `to_python` method that knows how to serialize the type safely.

**classmethod cast**(*value*)
Casts the attribute value as the defined __base_type__.

**classmethod get_base_type**()
Returns the __base_type__

**to_python**(*value*, *simple=False*)
Returns a json-safe, serialiazed version of the attribute

**to_string**(*value*)
Utility method that knows how to safely convert the value into a string

**class** `repocket.attributes.`**`AutoUUID`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Automatically assigns a uuid1 as the value. `__base_type__ = uuid.UUID`

**class** `repocket.attributes.`**`ByteStream`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Handles bytes that will be stored as a string in redis `__base_type__ = bytes`

**class** `repocket.attributes.`**`Bytes`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Handles raw byte strings `__base_type__ = bytes`

**class** `repocket.attributes.`**`DateTime`**(*auto_now=False*, *null=False*)
Repocket treats its models and attributes as fully serializable. Every attribute contains a `to_python` method that knows how to serialize the type safely.

**class** `repocket.attributes.`**`Decimal`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Handles Decimal `__base_type__ = Decimal`

**class** `repocket.attributes.`**`Float`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Handles float `__base_type__ = float`

**class** `repocket.attributes.`**`Integer`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Handles int `__base_type__ = int`

**class** `repocket.attributes.`**`JSON`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
This special attribute automatically stores python data as JSON string inside of redis. ANd automatically deserializes it when retrieving. `__base_type__ = unicode`

**class** `repocket.attributes.`**`Pointer`**(*to_model*, *null=False*)
Think of it as a soft foreign key.

This will automatically store the unique id of the target model and automatically retrieves it for you.

**classmethod** **`cast`**(*value*)
this method uses a redis connection to retrieve the referenced item

**class** `repocket.attributes.`**`UUID`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Automatically assigns a uuid1 as the value. `__base_type__ = uuid.UUID`

**class** `repocket.attributes.`**`Unicode`**(*null=False*, *default=None*, *encoding=u'utf-8'*)
Handles unicode-safe values `__base_type__ = unicode`

## 4.2 Redis connections

**class** `repocket.connections.`**`configure`**
global redis connection manager. this class is intended to be used as a singleton:

- the `connection_pool` method will set a global connection pool with the given `hostname`, `port` and `db`
- the `get_connection` can be used safely at any time after `connection_pool` was already set.

**classmethod** **`connection_pool`**(*hostname='localhost'*, *port=6379*, *db=0*)
sets the global redis connection pool.

**arguments**

- `hostname` - a string pointing to a valid hostname, defaults to `localhost`
- `port` - an integer with the port to connect to, defaults to `6379`
- `db` - a positive integer with the redis db to connect to, defaults to `0`

**classmethod** **`get_connection`**()
returns a connection from the pool. this method should **only** be called after you already called `connection_pool`

## 4.3 Models

**class** `repocket.model.`**`ActiveRecord`**(**args*, ***kw*)
base model class, this is how you declare your active record.

```
class User(ActiveRecord):
    id = attributes.AutoUUID()
    github_access_token = attributes.Bytes()
    name = attributes.Unicode()
    email = attributes.Unicode()
    carpentry_token = attributes.Bytes()
    github_metadata = attributes.JSON()
```

(continues on next page)

(continued from previous page)

```
obj1 = User(
    github_access_token=b'sometoken',
    email='foo@bar.com',
    carpentry_token=b'1234',
    github_metadata={
        'yay': 'this is json baby!'
    }
)

key = obj1.save()
connection = configure.get_connection()
raw_results = connection.hgetall(key)
```

**classmethod create**(***kwargs*)
: Takes all the valid attributes of an active record, saves it immediately and returns the instance, ready for further manipulation.

**delete**()
: Deletes all the redis keys used by this model

**matches**(*kw*)
: Takes a dictionary with keyword args and returns true if all the args match the model field values

**save**()
: Persists the model in redis. Automatically generates a primary key value if one was not provided

## 4.4 Exceptions

**exception** repocket.errors.**RepocketActiveRecordDefinitionError**
: Exception raised when a model has more than one AutoUUID or any other kind of inconsistency in the model declaration.

# CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index