Coralillo Documentation

Release 1.8.0

Abraham Toriz Cruz

Feb 18, 2021

Contents

1 Installation **3**

2 Basic Usage **5**

3 Learn More **7**
- 3.1 Connection parameters . . . 7
- 3.2 Fields . . . 8
- 3.3 Validation . . . 9
- 3.4 Flask Integration . . . 11
- 3.5 Lua scripting . . . 12
- 3.6 Multi-tenancy . . . 13
- 3.7 Atomic operations . . . 13
- 3.8 Extending . . . 14
- 3.9 Design desitions . . . 14
- 3.10 Helpers . . . 14
- 3.11 API . . . 14

4 Indices and tables **15**

Index **17**

Coralillo is an ORM (Object-Redis Mapping) for pyton. It is named after a little red snake (Coral snake) that you can find in México.

CHAPTER 1

Installation

Install it via `pip`

```
$ pip install coralillo
```

It is good idea to manage your dependencies inside a virtualenv.

CHAPTER 2

Basic Usage

```
from coralillo import Engine, Model, fields

# Create the engine
eng = Engine()

# Declare your models
class User(Model):
    name      = fields.Text()
    last_name = fields.Text()
    email     = fields.Text(
        index=True,
        regex='^[\w.%+-]+@[\w.-]+\.[a-zA-Z]{2,}$',
    )

    class Meta:
        engine = eng

# Persist objects to database
john = User(
    name='John',
    last_name='Doe',
    email='john@example.com',
).save()

# Query by index
mary = User.get_by('email', 'mary@example.com')

# Retrieve all objects
users = User.all()
```

CHAPTER 3

Learn More

3.1 Connection parameters

By default `Engine()` connects to localhost using the default port and database number **0**. If you want to connect to a different host, port or database you can use an URL like in the following example:

```
from coralillo import Engine

HOST = 'localhost'
PORT = 6379
DB = 0

redis_url = 'redis://{host}:{port}/{db}'.format(
   host = HOST,
   port = PORT,
   db = DB,
)
eng = Engine(url=redis_url)
```

For more information on how to build the URL refer to https://github.com/andymccurdy/redis-py/blob/master/redis/client.py#L462 .

Another option would be to pass the configuration parameters directly like this:

```
from coralillo import Engine

HOST = 'localhost'
PORT = 6379
DB = 0

eng = Engine(
   host = HOST,
   port = PORT,
   db = DB,
)
```

For a full reference on the keyword arguments that you can pass refer to https://github.com/andymccurdy/redis-py/blob/master/redis/client.py#L490 .

3.2 Fields

Fields let you define your object's properties and transform the values retrieved from the database, we support the following:

- `fields.Text` A simple text field
- `fields.Hash` A hashed text using bcrypt
- `fields.Bool` A true/false value
- `fields.Integer` An integer
- `fields.Float` A floating point value
- `fields.Datetime` A date and time
- `fields.Location` A pair of latitude/longitude

3.2.1 Relation fields

We also provide fields for defining relationships with other models in a ORM-fashion

- `fields.SetRelation` Stored as a set of the related ids
- `fields.SortedSetRelation` Stored as a sorteed set of the related ids, using a sotring key
- `fields.ForeignIdRelation` simply stores the string id of the related object

3.2.2 Indexes

Only Text fields are ready to be indexes

3.2.3 Creating your own fields

Simply subclass `Field` or `Relation`.

NORM fields follow an specific workflow to read/write from/to the redis database. Such workflow needs the following methods to be implemented (or inherited) for each field:

- `__init__` for field initialization, don't forget to call the parent's constructor
- `init` is called to parse a value given in the model's constructor
- `recover` is called to parse a value retrieved from database
- `prepare` is called to transform values or *prepare* them to be sent to database
- `to_json` should return the json-friendly version of the value
- `validate` is called when doing `Model.validate(data)` or `obj.update(data)`

Additionally, the following methods are needed for `Relation` subclasses:

save(*value*, *pipeline*[, *commit=True*])
persists this relationship to the database

relate(*obj*, *pipeline*)
sets the given object as related to the one that owns this field

delete(*pipeline*)
tells what to do when a model with relationships is deleted

key()
returns a fully qualified redis key to this relationship

get_related_ids()
for subclasses of `SetRelation`, returns the list of related ids

fill()
is called when you need to know the relationships for a model. Usually via the proxy object.

__contains__(*obj*)
is for subclasses of `SetRelation` and should tell wether or not the given object is in this relation. Usually called via the proxy object.

3.3 Validation

Coralillo includes validation capabilities so you can check the data sent by a request before creating an object.

Validation code is part of the `coralillo.Form` class, which is parent of the `coralillo.Model` class.

3.3.1 Basic usage

In its simplest form, validation ensures that the data passed to the validation function matches the field definition of the class:

```
from coralillo import Form, Engine, fields, errors

eng = Engine()

class MyForm(Form):
   field1 = fields.Text()
   field2 = fields.Text(required=False)

   class Meta:
      engine = eng

try:
   MyForm.validate()
except errors.ValidationErrors as ve:
   assert len(ve) == 1
   assert ve[0].field == 'field1'

data = MyForm.validate(field1='querétaro', field2='chihuahua')

assert data.field1 == 'querétaro'
assert data.field2 == 'chihuahua'
```

3.3.2 Default validations

Validation rules are built on field definition, here are some rules that are automatically added in addition to `required` rule.

```
from coralillo import Model, Engine, fields, errors

eng = Engine()

class Base(Model):

   class Meta:
      engine = eng

# Validate uniqueness of indexes
class Uniqueness(Base):
   username = fields.Text(index=True)

Uniqueness(username='foo').save()

try:
   Uniqueness.validate(username='foo')
except errors.ValidationErrors as ve:
   assert isinstance(ve[0], errors.NotUniqueFieldError)

# Validate regexes
class Regex(Base):
   css_color = fields.Text(regex=r'#[0-9a-f]{6}')

try:
   Regex.validate(css_color='white')
except errors.ValidationErrors as ve:
   assert isinstance(ve[0], errors.InvalidFieldError)

# Validate forbidden values
class Forbidden(Base):
   name = fields.Text(forbidden=['john'])

try:
   Forbidden.validate(name='john')
except errors.ValidationErrors as ve:
   assert isinstance(ve[0], errors.ReservedFieldError)

# Validate allowed values
class Allowed(Base):
   name = fields.Text(allowed=['john'])

try:
   Allowed.validate(name='maría')
except errors.ValidationErrors as ve:
   assert isinstance(ve[0], errors.InvalidFieldError)
```

3.3.3 Non fillable fields

Sometimes you might want to prevent a field from being filled or validated using the `coralillo.Form.validate()`, in that case the keyword argument `fillable` of a field will do the trick.

```
from coralillo import Form, Engine, fields, errors

eng = Engine()

class MyForm(Form):
   field1 = fields.Text(fillable=False)

   class Meta:
      engine = eng

data = MyForm.validate(field1='de')

assert data.field1 is None
```

3.3.4 Custom rules

You can add custom rules to your forms or models to make even more complicated validation rules. Simply apply the `coralillo.validation.validation_rule()` decorator to a function in your class and write your code so that it raises the appropiate subclass of `coralillo.errors.BadField` as shown in the example.

```
from coralillo import Form, Engine, fields, errors
from coralillo.validation import validation_rule

eng = Engine()

class Myform(Form):
   password = fields.Text()
   confirmation = fields.Text()

   @validation_rule
   def confirmation_matches(data):
      if data.password != data.confirmation:
         raise errors.InvalidFieldError(field='confirmation')

try:
   MyForm.validate(password='foo', confirmation='var')
except errors.ValidationErrors as ve:
   assert ve[0].field == 'confirmation'
```

3.4 Flask Integration

There is a module for that!

```
$ pip instal flask-coralillo
```

The following example creates a simple flask application that creates and lists objects.

```
# app.py
from flask import Flask, request, redirect
from flask_coralillo import Coralillo
from coralillo import Model, fields

app = Flask(__name__)
```

(continues on next page)

(continued from previous page)

```
engine = Coralillo(app)

class Car(Model):
    name = fields.Text()

    class Meta:
        engine = engine

@app.route('/')
def list_cars():
    res = '<h1>Cars</h1><ul>'

    for car in Car.get_all():
        res += '<li>{}</li>'.format(car.name)

    res += '</ul><h3>Add car</h3>' + \
        '<form method="POST">' + \
        '<input name="name">' + \
        '<input type="submit" value="Add">' + \
        '</form>'

    return res

@app.route('/', methods=['POST'])
def add_car():
    newcar = Car.validate(**request.form.to_dict()).save()

    return redirect('/')

if __name__ == '__main__':
    app.run()
```

Now if you run `python app.py` and you visit `http://localhost:5000` you will be able to intercact with your brand new Flask-Coralillo application.

3.5 Lua scripting

Coralillo uses a few lua scripts to atomically run certain operations. These can be accessed through the engine's `lua` object. Here are the available scripts:

`engine.lua.`**`drop`**(*args=[pattern]*)
: Deletes all keys matching `pattern` from the database. Specially useful in tests.

`engine.lua.`**`allow`**(*args=[objspec], keys=[allow_key]*)
: Adds objspec to the permission tree stored at `allow_key`

3.5.1 Script registering

You can add your own scripts using Coralillo's lua interface like this:

```
from coralillo import Engine

eng = Engine()
```

(continues on next page)

(continued from previous page)

```
script = 'return ARGV[1]'

eng.lua.register('my_script', script)

assert eng.lua.my_script(args=['hello']) == b'hello'
```

Lua.**register**(*scriptname*, *scriptbody*)
Registers script defined by `scriptbody` (a string) so it is accessible through the `lua` interface of the engine under the name `scriptname`.

3.6 Multi-tenancy

It is often useful to have objects of the same class stored within different namespaces, for example when running an application that serves different clients and you don't want them to be in the same place.

For this case Coralillo has a Model subclass called BoundedModel that lets you specify a prefix for your models:

```
from coralillo import Engine, BoundedModel, fields

eng = Engine()

current_namespace = 'coral'

class User(BoundedModel):
    name = fields.Text()

    @classmethod
    def prefix(cls):
        # here you may have your own way of determining the __bound__
        # depending on the context. We will just return a variable's
        # value
        return current_namespace

    class Meta:
        engine = eng

# models are saved in the namespace given by the context
juan = User(name='Juan').save()
assert eng.redis.exists('coral:user:members')

# changing the context changes how models are found
current_namespace = 'nauyaca'
assert User.get(juan.id) is None

pepe = User(name='Pepe').save()
assert eng.redis.exists('nauyaca:user:members')
```

3.7 Atomic operations

Describe which of the operations are done atomically

3.8 Extending

How to extend

3.9 Design desitions

for example why table names are singular

3.10 Helpers

Currentry three helpers exist:

3.11 API

CHAPTER 4

Indices and tables

- genindex
- search

Index

Symbols

__contains__() (*built-in function*), 9

D

delete() (*built-in function*), 9

E

engine.lua.allow() (*built-in function*), 12
engine.lua.drop() (*built-in function*), 12

F

fill() (*built-in function*), 9

G

get_related_ids() (*built-in function*), 9

K

key() (*built-in function*), 9

L

Lua.register() (*built-in function*), 13

R

relate() (*built-in function*), 8

S

save() (*built-in function*), 8