morbo Documentation

Release 0.9

Elisha Cook

August 29, 2014

Contents

Morbo is another python package for mapping objects to mongodb. Its goal is to provide a friendly API that includes validation and relationships without enforcing concepts from relational databases or obscuring the already very nice pymongo query interface. There are lots of existing packages for this sort of task. Some popular ones are MongoKit, MongoEngine and Ming. Morbo was written partly because no existing package meets all of the above goals and partly because writing it is fun.

CHAPTER 1

Example

```
from morbo import *

class Person(Model):
        name = Text(required=True, maxlength=100)
        email = Email(required=True)


class Recipe(Model):
        name = Text(required=True, maxlength=100)
        author = One(Person)
        ingredients = ManyToMany('Ingredient', inverse='recipes')
        instructions = Text()


class Ingredient(Model):
        name = Text(required=True, maxlength=100)


connection.setup('morbo_recipe_box')

bob = Person(name='Chef Bob', email="bob-the-chef@example.com")
bob.save()

recipe = Recipe(name="Cinnamon & Sugar Popcorn")
recipe.save()
recipe.author = bob

for n in ('popcorn', 'coconut oil', 'sugar', 'cinnamon'):
        ingredient = Ingredient(name=n)
        ingredient.save()
        recipe.ingredients.add(recipe)

cinnamon = Ingredients.find_one({'name':'cinnamon'})
recipe = cinnamon.recipes.find_one()
print "%s by %s" % (recipe.name, recipe.author.name)
```

1.1 Features

1.2 Modules

1.2.1 morbo.connection

1.2.2 morbo.validators

This package provides a simple validation system as well as a number of validators and converters. You'll probably find it's usage familiar.

Validators generally perform one task: check that a value meets a certain specification. Sometimes they also perform a conversion. For instance, a time validator might validate that a value is in one of a number of formats and then convert the value to, say, a datetime instance or a POSIX timestamp.

This package tries to supply a number of flexible validators but doesn't even come close to the kitchen sink. Most applications will have their own validation requirements and the thinking here is to make it as easy as possible to create new validators.

So, creating validators is straightforward. Subclass Validator and implement the validate() method. That's all that's required. Here is an example that will only validate yummy things. Optionally, it will convert to yumminess level:

```
class Yummy(Validator):
    yumminess = {
        'Pizza': 1.5,
        'Pie': 2.75,
        'Steak': 5.58,
        'Sushi': 14.62,
        'Duck Confit': 28.06
    }

    def __init__(self, should_convert=False, **kwargs):
        self.should_convert = should_convert
        super(Yummy, self).__init__(**kwargs)


    def validate(self, value):
        if value not in self.yumminess:
            raise InvalidError('Yumminess not known for "%s"' % value)

        if self.should_convert:
            return self.yumminess[value]

        return value
```

It's a convention, but not a requirement, to put error values in the class like so:

```
class Yummy(Validator):
    NOT_YUMMY = "This is not yummy"

    def validate(self, value):
        if not self.is_yummy(value):
            raise InvalidError(self.NOT_YUMMY)
```

Then we can do things based on the type of error, if we so desire:

```
yummy_validator = Yummy()
try:
    yummy_validator.validate('Fried Okra')
    print "Fried Okra is yummy!"
except InvalidError, e:
    if e.message == Yummy.NOT_YUMMY:
        print "Fried Okra is not yummy"
    else:
        print "There is something wront with Fried Okra"
```

exception morbo.validators.**InvalidError**

This exception is thrown by all validators to indicate that data is invalid:

```
v = SomeValidator()
try:
    v.validate("puppies")
except InvalidError, e:
    print "Oh no! Something's wrong! %s" % e.message
```

exception morbo.validators.**InvalidGroupError**(*errors*)

This exception represents a group of invalid errors. It takes a dict where the keys are, presumably, the names of items that were invalid and the values are the errors for their respective items:

```
def check_stuff_out(stuff):
    validator = SomeValidator()
    errors = {}

    for k,v in stuff.items():
        try:
            validator.validate(v)
        except InvalidError, e:
            errors[k] = e

    if errors:
        raise InvalidGroupError(errors)
```

class morbo.validators.**Validator**(*optional=False*, *default_value=None*)

This is the base validator class. Don't use it unless you are subclassing to create your own validator.

class morbo.validators.**GroupValidator**(***kwargs*)

Validates a a dict of *key => validator*:

```
v = GroupValidator(foo=Text(), bar=TypeOf(int))
v.validate({'foo':'oof', 'bar':23}) # ok
v.validate(5) # nope
v.validate('foo':'gobot', 'bar':'pizza') # nope

# unspecified keys are filtered
v.validate({'foo': 'a', 'goo': 'b', 'bar':17})
# ok -> {'foo': 'a', 'bar':17}

# optional validators are, well, optional
v = GroupValidator(foo=Text(), bar=TypeOf(int, optional=True, default_value=8))
v.validate({'foo':'ice cream'}) # -> {'foo':'ice cream', 'bar': 8}
```

class morbo.validators.**Text**(*minlength=None*, *maxlength=None*, ***kwargs*)

Passes text, optionally checking length:

```
v = Text(minlength=2,maxlength=7)
v.validate("foo") # ok
v.validate(23) # oops
v.validate("apron hats") # oops
v.validate("f") # oops
```

class morbo.validators.**Email**(*optional=False*, *default_value=None*)
Passes email addresses that meet the guidelines in RFC 3696:

```
v = Email()
v.validate("foo@example.com") # ok
v.validate("foo.bar_^&!baz@example.com") # ok
v.validate("@example.com") # oops!
```

class morbo.validators.**DateTime**(*default_format='%x %X'*, *use_timelib=True*, *use_dateutil=True*, ***kwargs*)
Validates many representations of date & time and converts to datetime.datetime. It will use timelib if available, next it will try dateutil.parser. If neither is found, it will use `datetime.strptime()` with some predefined format string. Int or float timestamps will also be accepted and converted:

```
# assuming we have timelib
v = DateTime()
v.validate("today") # datetime.datetime(2011, 9, 17, 0, 0, 0)
v.validate("12:06am") # datetime.datetime(2011, 9, 17, 0, 6)
v.validate(datetime.now()) # datetime.datetime(2011, 9, 17, 0, 7)
v.validate(1316232496.342259) # datetime.datetime(2011, 9, 17, 4, 8, 16, 342259)
v.validate("baloon torches") # oops!
```

class morbo.validators.**Bool**(*optional=False*, *default_value=None*)
Passes and converts most representations of True and False:

```
b = Bool()
b.validate("true") # True
b.validate(1) # True
b.validate("yes") # True
b.validate(True) # True
b.validate("false") # False, etc.
```

class morbo.validators.**BoundingBox**(*optional=False*, *default_value=None*)
Passes a geographical bounding box of the form SWNE, e.g., 37.73,-122.48,37.78,-122.37. It will accept a list or a comma-separated string:

```
v = BoundingBox()
b.validate([42.75804,-85.0031, 42.76409, -84.9861]) # ok
b.validate("42.75804,-85.0031, 42.76409, -84.9861") # -> [42.75804,-85.0031, 42.76409, -84.9861]
```

class morbo.validators.**LatLng**(*optional=False*, *default_value=None*)
Passes a geographical point in for form of a list, tuple or comma-separated string:

```
v = LatLng()
v.validate("42.76066, -84.9929") # ok -> (42.76066, -84.9929)
v.validate((42.76066, -84.9929)) # ok
v.validate("234,56756.453") # oops
```

class morbo.validators.**Enum**(**values*, ***kwargs*)
Passes anything that evaluates equal to one of a list of values:

```
v = Enum('a', 'b', 'c')
v.validate('a') # ok
v.validate('d') # nope!
```

class morbo.validators.**TypeOf**(**types*, ***kwargs*)
Passes any value of a specified type:

```
v = TypeOf(float)
v.validate(0.4) # ok
v.validate(1) # nope

# more than one type is ok too
v = TypeOf(int, float, complex)
v.validate(5) # ok
v.validate(5.5) # ok
v.validate(complex(5,5)) # ok
```

class morbo.validators.**URL**(***kwargs*)
Passes a URL using guidelines from RFC 3696:

```
v = URL()
v.validate('http://www.example.com') # ok
v.validate('https://www.example.com:8000/foo/bar?smelly_ones=true#dsfg') # ok
v.validate('http://www.example.com/foo;foo') # nope

# You can also set which schemes to match
v = URL(schemes=('gopher',))
v.validate('gopher://example.com/') # ok
v.validate('http://example.com/') # nope!
```

Regex from http://daringfireball.net/2010/07/improved_regex_for_matching_urls.

class morbo.validators.**OneOf**(**validators*, ***kwargs*)
Passes values that pass one of a list of validators:

```
v = OneOf(URL(), Enum('a', 'b'))
v.validate('b') # ok
v.validate('http://www.example.com') # ok
v.validate(23) # nope
```

class morbo.validators.**ListOf**(*validator*, ***kwargs*)
Passes a list of values that pass a validator:

```
v = ListOf(TypeOf(int))
v.validate([1,2,3]) # ok
v.validate([1,2,"3"]) # nope
v.validate(1) # nope
```

class morbo.validators.**Anything**(*optional=False*, *default_value=None*)
Passes anything

1.2.3 morbo.model

Models are classes whose instances persist in a MongoDB database. Other than the special validator fields, Models are just regular python classes. You can have whatever non-validator attributes and methods you want. Along with simple persistence, Model subclasses also provide methods for finding, removing and updating instances using almost the same API as pymongo.

class morbo.model.**Model**(***kwargs*)
A class with persistent instances:

```
class Swallow(Model):
    is_laden = Bool()
    speed = TypeOf(float)
    direction = Enum('N', 'E', 'S', 'W')

s = Swallow(is_laden=True, speed=23.7, direction='E')
s.save()
repr(s._id) # ObjectId('4e7be644d761504a98000000')
```

classmethod count()
Get a count of the total number of instances of this model.

classmethod find(*spec=None*, **args*, ***kwargs*)
Find instances of this model. This method is exactly the same as pymongo.Collection.find() except that the returned cursor returns instances of this model instead of dicts:

```
class Swallow(Model):
    ...

s = Swallow(is_laden=True, speed=23.7, direction='E')
s.save()

Swallow.find({'direction':'E'}).next() # <Swallow 4e7bea8fd761504ae3000000>
```

classmethod find_one(*spec_or_id=None*, **args*, ***kwargs*)
Just like pymongo.Collection.find_one() but returns an instance of this model.

classmethod get_collection()
Get a pymongo.Collection instance for the collection that backs instances of this model:

```
class Swallow(Model):
    ...

Swallow.get_collection() # Collection(Database(Connection('localhost', 27017), u'morbotests'
```

classmethod remove(*spec=None*)
Works just like pymongo.Collection.remove(). Note that this is different than the model instance method remove() which removes just that single instance.

save()
Save this instance to the database. Validation is performed first.

validate()
Run validation on all validated fields.

was_created()
Called after the model is saved for the first time.

was_modified()
Called each time the model is saved after the first save.

1.2.4 morbo.relationships

CHAPTER 2

Index

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

m