Simon Documentation

Release 0.5.0

Andy Dirnberger

March 19, 2013

CONTENTS

Meet Simon. He wants to help you create simple MongoDB models.

CHAPTER

ONE

OVERVIEW

Simon is a model library for MongoDB. It aims to introduce the features of MongoDB and PyMongo in a Pythonic way. It allows you to work with objects and methods instead of only allowing you to work with everything as a `dict`.

Simon tries emphasize the flexibility and power of MongoDB. It does this in a couple of ways. First, unlike other libraries of its kind, Simon does not enforce a schema on your documents. This allows you to take advantage of the dynamic schemas offered by MongoDB. Second, while Simon allows you to perform traditional saves, it also allows you full control over performing atomic updates. This is covered in greater detail in the *Basic Usage* section.

1.1 Quickstart

Ready to dive in? Here's a quick run through the basics of using Simon. It will guide you through defining models, saving and retrieving documents, and connecting to a database.

1.1.1 Defining a Model

To define a simple model, all you need to do is inherit from the `Model` class.

```
from simon import Model

class User(Model):
    """This is the model used for users."""
```

This will define the `User` model which will use the `users` collection in the database.

1.1.2 Using the Model

To instantiate a new `User`:

```
user = User(name='Simon', email='simon@example.com')
```

Attributes can also be assigned after instantiation.

```
user = User()
user.name = 'Simon'
user.email = 'simon@example.com'
```

1.1.3 Saving

Saving the changes is as easy as calling `save()`. `created` and `modified` dates will be added to the document before it is written to the database, and the `ObjectId` assigned by the database will be added to the instance. (`created` will only be added to documents that haven't already been saved and don't already have a `created` field.)

```
user.save()

print '%r %r %r' % (user.id, user.created, user.modified)
# ObjectId('50e467580ea5faf0b83679f7') datetime.datetime(2013, 1, 2, 16, 59, 4, 688000) datetime.date
```

By default saves do not happen with write concern set. There is no guarantee the document will make it to the database. Write concern can be turned on by setting the `safe` parameter to `True`.

```
user.save(safe=True)
```

1.1.4 Retrieving

Once the document has been saved it can easily be retrieved from the database. The `get()` method accepts the names of fields as parameters with values to match against.

```
user = User.get(name='Simon')
print '%r %r' % (user.name, user.email)
# 'Simon' 'simon@example.com'
```

For information about the possible exceptions associated with `get()`, check out Exceptions.

Retrieving multiple documents instead of just one is also easy. Just use the `find()` method instead of `get()`. They accept parameters the same way.

```
user2 = User(name='Simon', email='simon@example.org')
user2.save()

users = User.find(name='simon')
for user in users:
    print '%r %r' % (user.name, user.email)

# 'Simon' 'simon@example.com'
# 'Simon' 'simon@example.org'
```

1.1.5 Connecting to a Database

Before you can use your models, you need to connect to a database. This is done by using the `connect()` method.

```
from simon.connection import connect

connect('localhost', name='simon')
```

This will open a connection to the `simon` database on `localhost`. It's also possible to connect to a database on a remote server.

```
connect('simon.example.com', name='simon')
```

Or you can specify a full URI.

```
connect('mongodb://simon.example.com/simon')
```

When connecting to a database that requires authentication, a username and password can be specified either through the `username` and `password` arguments or as part of the URI.

```
connect('localhost', name='simon', username='user', password='passwd')

# ~ or ~

connect('mongodb://user:passwd@simon.example.com/simon')
```

1.1.6 Exceptions

When using the `get()` method from a model class it is important to keep in mind that there are a couple of exceptions it can raise. It's a good idea to catch them.

```
try:
    user = User.get(name='Simon2')
except User.NoDocumentFound:
    # This means no documents matched the query
    handle_the_exception()

try:
    user = User.get(name='Simon')
except User.MultipleDocumentsFound:
    # This means more than one document matched the query
    handle_the_exception()
```

There is also an exception that can be raised when connecting to a database.

```
try:
    connect('locahost', name='simon')
except ConnectionError:
    # There was a problem connecting to the database
    handle_the_exception()
```

1.2 Basic Usage

Simon offers a lot of flexibility in how you interact with the database.

All of the examples below utilize the `User` model defined in *Quickstart*, so if you haven't already done that, you want to check it out first.

1.2.1 Retrieving

At the heart of retrieving documents are three methods: `all()`, `find()`, and `get()`.

Use `all()` to retrieve all documents from the `users` collection.

```
users = Users.all()
```

Often times it will be necessary to filter the documents coming back. To do so, use `find()`. It takes a series of named parameters that represent keys in the documents and the values to match against.

To find all documents whose `name` field has a value of `Simon`:

```
users = Users.find(name='Simon')
```

To find all documents whose `name` field has a value of `Simon` and whose `company` field has a value of `My Company`:

```
internal_users = Users.find(name='Simon', company='My Company')
```

If you were to execute these queries using the mongo Shell, they would look like:

```
users = db.users.find({name: 'Simon'})

internal_users = db.users.find({name: 'Simon', company: 'My Company'})
```

At this point, no real information would have been returned from the database. Utilizing the cursor behavior built into PyMongo, documents will only be transferred from the database when they are requested. This is done by interacting with the result of `find()` like you would with any other iterable such as a `list`.

```
for user in users:
    print 'A document was just loaded from the users collection'
```

Documents can also be loaded through slicing, although this will cause all documents in, as well as prior to, the slice to be loaded.

```
first_user = users[0]
# the first user has been loaded

fourth_users = users[3]
# the first four users have been loaded

all_users = users[:]
# all users have been loaded
```

More advanced uses are covered in *Querying*.

1.2.2 Saving

The main way to save a document using Simon is with `save()`. Calling it on an instance with a new document will insert the document. The document will be given an `ObjectId` by the database, which will then be associated with the instance.

```
user = User(name='Simon')
user.save()  # insert
```

Calling `save()` on an instance with an existing document will update the document. This will replace what's in the database with the one associated with the instance.

```
user.email = 'simon@example.org'
user.save()  # update
```

The equivalent queries in the mongo Shell would be:

```
db.users.insert({name: 'Simon'})

db.users.update({_id: ObjectId(...)}, {email: 'simon@example.org'})
```

More advanced uses are covered in *Saving*.

1.2.3 Deleting

If you don't want a document anymore, removing it from the database is simply a matter of calling `delete()`.

```
user.delete()
```

Be careful as this will raise a `TypeError` if you try to delete a document that was never saved.

If you were to execute this query directly in mongo Shell, it would look like:

```
db.users.remove({_id: ObjectId(...)})
```

At the time of this writing there appears to be no way to set the `justOne` parameter to `true` using `PyMongo`. If you decide to remove the unique constraint from the `_id` field, bad things could happen when you use `delete()`.

1.3 The QuerySet Class

Unlike `get()`, `all()` and `find()` return an instance of `QuerySet`. The `QuerySet` class utilizes PyMongo `cursors` to limit the amount of data that is actually transferred from the database.

Additionally it also exposes a few additional methods for controlling the database that is returned.

Full documentation for `QuerySet` is available in *Simon API*.

1.3.1 Sorting

Instances of `QuerySet` can be sorted through the `sort()` method. It is called by passing in a series of field names, each one optionally prefixed by a − to denote that the field should be sorted in descending order.

If you sort by a field that doesn't exist in all documents, a document without the field will be treated as if it has a value less than that of a document that has the field.

```
# sort by name ascending
users = User.all().sort('name')
```

Sorting by multiple fields is just as easy.

```
# sort by name and email ascending
users = User.all().sort('name', 'email')

# sort by name ascending and email descending
users = User.all().sort('name', '-email')
```

When sorting by multiple fields, the direction of one field's sort will not affect the direction of other sorts.

```
# sort by name ascending, email descending, and date_of_birth ascending
users = User.all().sort('name', '-email', 'date_of_birth')
```

Here are the queries in the mongo Shell:

```
users = db.users.find().sort({name: 1})

users = db.users.find().sort({name: 1, email: 1})

users = db.users.find().sort({name: 1, email: -1})

users = db.users.find().sort({name: 1, email: -1, date_of_birth: 1})
```

1.3.2 Limiting

When querying for documents, you may only want a subset of the documents that match your query. Simon allows you to control this through two methods, `limit()` and `skip()`. These allow you to control the number of documents returned and the number of documents to omit.

```
# retrieve the first 10 documents
users = User.all().limit(10)

# skip the first 10 documents
users = User.all().skip(10)
```

`limit()` and `skip()` can be chained together to create paged results.

```
# retrieve the second page of 10 documents
users = User.all().limit(10).skip(10)
```

The methods can be used in any order.

```
# retrieve the second page of 10 documents
users = User.all().skip(10).limit(10)
```

Here are the queries in the mongo Shell:

```
users = db.users.find().limit(10)

users = db.users.find().skip(10)

users = db.users.find().limit(10).skip(10)

users = db.users.find().skip(10).limit(10)
```

1.3.3 Distinct

It is possible to get a list of unique values for a single field using `distinct()`.

```
# get a list of all email addresses for users named Simon
emails = User.find(name='Simon').distinct('email')
```

Unlike Simon, the same query in the mongo Shell is handled at the collection level:

```
names = db.users.distinct('email', {name: 'Simon'})
```

1.3.4 Length

Sometimes all you need is how many documents match your query. Simon provides that information in `count()`.

```
count = User.all().count()
```

Simon makes sure that the any calls to `limit()` and `skip()` are factored in. Executing the same thing in mongo Shell would look like:

```
count = db.users.find().count(true)
```

1.4 Querying

MongoDB offers a lot of flexibility when querying for documents in the database. Simon tries to expose that flexibility in easy to use ways.

To use one of MongoDB's operators, append it to the name of the field you want to apply it to, separated by a double underscore (__). Simon will automatically translate it into the correct query syntax.

```
# match users where name is not equal to 'Simon'
users = User.find(name__ne='Simon')

# match users where score is greater than 1000
users = User.find(score__gt=1000)

# match users created in 2012
from datetime import datetime
jan_1_2012 = datetime(2012, 1, 1)
jan_1_2013 = datetime(2013, 1, 1)
users = User.find(created__gte=jan_1_2012, created__lt=jan_1_2013)
```

There's queries will be translated to:

```
users = db.users.find({name: {$ne: 'Simon'}})

users = db.users.find({score: {$gt: 1000}})

jan_1_2012 = new Date(2012, 1, 1)
jan_1_2013 = new Date(2013, 1, 1)
users = db.users.find({created: {$gte: jan_1_2012, $lt: jan_1_2013}})
```

More information about all of the operators offered by MongoDB is available in the MongoDB docs.

1.4.1 Comparison Operators

The full list of comparison operators available is:

gt Matches documents where the field's value is greater than the specified value.

```
users = User.find(score__gt=1000)
```

gte Matches documents where the field's value is greater than or equal to the specified value.

```
users = User.find(score__gte=1000)
```

lt Matches documents where the field's value is less than the specified value.

```
users = User.find(score__lt=1000)
```

lte Matches documents where the field's value is less than or equal to the specified value.

```
users = User.find(score__lte=1000)
```

ne Matches documents where the field's value is not equal to the specified value.

```
users = User.find(name__ne='Simon')
```

in Matches documents where the field's value is equal to any of the values in the specified list.

```
users = User.find(name__in=['Alvin', 'Simon', 'Theodore'])
```

nin Matches documents where the field's value is not equal to any of the values in the specified list.

```
users = User.find(name__nin=['Alvin', 'Simon', 'Theodore'])
```

all Matches documents where the field holds a list containing all of the specified elements.

```
users = User.find(friends__all=['Alvin', 'Theodore'])
```

1.4.2 Element Operators

The full list of element operators available is:

exists Matches documents where the field's existence matches the specified value.

```
users = User.find(email__exists=True)
```

1.4.3 Array Operators

The full list of array operators available is:

elemMatch Matches documents where the field is a list matching the specified query.

```
users = User.find(addresses__elemMatch={'state': 'NY'})
```

size Matches documents where the field is a list of the specified length.

```
users = User.find(fields__size=2)
```

1.4.4 Geospatial Operators

One of the most powerful ways to query with MongoDB is through geospatial querying. Unlike the operators discussed thus far, Simon exposes the geospatial operators through convenience methods that help harness the full potential of each operator.

Before you can use any of these operators, you will need to create a `two-dimensional index`.

```
db.users.ensureIndex({location: '2d'})
```

The convenience methods can be used by importing the `geo` module.

```
from simon import geo
```

near Matches documents from nearest to farthest with respect to the specified point.

```
users = User.find(location=geo.near([x, y]))
```

within Matches documents contained within the specified shape.

```
users = User.find(location=geo.within('box', [x1, y1], [x2, y2]))
```

While `within()` can be used on its own, the following methods make it even easier.

box Matches documents within the specified rectangular shape.

```
users = User.find(location=geo.box([x1, y1], [x2, y2]))
```

polygon Matches documents within the specified polygonal shape.

```
users = User.find(location=geo.polygon([x1, y1], [x2, y2], [x3, y3]))
```

center Matches documents within the specified circular shape. **Note** the `center` operator is accessed through the `circle()` method.

```
center = [x, y]
users = User.find(location=geo.circle(center, radius))
```

Here's a quick run through of these queries in the mongo Shell:

```
users = db.users.find({location: {$near: [x, y]}})

users = db.users.find({location: {$within: {$box: [[x1, y1], [x2, y2]]}}})

users = db.users.find({location: {$within: {$polygon: [[x1, y1], [x2, y2], [x3, y3]]}}})

users = db.users.find({location: {$within: {$center: [[x, y], radius]}}})
```

The full list of options offered by each method can be found in the *Geo* section of *Simon API*.

1.4.5 Logical Operators

Sometimes more complex queries require combining conditions with logical operators, such as `AND`, `OR`, and `NOT`.

not Performs a logical `NOT` operation on the specified expression.

```
users = User.find(score__not__gt=1000)
```

To perform this query in the mongo Shell:

```
users = db.users.find({score: {$not: {$gt: 1000}}})
```

Using the `AND` and `OR` operators with Simon requires the assistance of `Q` objects. Fortunately they work just like any other query with Simon. Instead of passing the the query directly to a method like `find()`, however, the query is passed to `Q`.

```
from simon.query import Q
query = Q(name='Simon')
```

The new object is then combined with one or more additional `Q` objects, the end result of which is then passed to `find()`. `Q` objects are combined using bitwise and (`&`) and or (`|`) to represent logical `AND` and `OR`, respectively.

```
# match users where name is equal to 'Simon' AND score is greater
# than 1000
users = User.find(Q(name='Simon') & Q(score__gt=1000))

# match users where name is equal to 'Simon' AND score is greater
# than 1000, OR name is either 'Alvin' or 'Theodore'
users = User.find(Q(name='Simon', score__gt=1000) | Q(name__in=['Alvin', 'Theodore']))

# match users who have no friends
users = User.find(Q(friends__exists=False) | Q(friends__size=0))
```

Any number of `Q` objects can be chained together. Be careful, however, as chaining together a lot of queries through different operators can result in deeply nested queries, which may become inefficient.

Here's how these queries would look in the mongo Shell:

```
users = db.users.find({$and: [{name: 'Simon'}, {score: {$gt: 1000}}]})

users = db.users.find({$or: [{name: 'Simon', score: {$gt: 1000}}, {name: {$in: ['Alvin', 'Theodore']

users = db.users.find({$or: [{friends: {$exists: false}}, {friends: {$size: 0}}]})
```

1.4.6 Exceptions

When using `get()` to retrieve a document, there are two potential exceptions that may be raised. When one of these exceptions is raised, it will be raised as part of the model class being queried.

Assume the following documents for all examples below.

`MultipleDocumentsFound` This exception is raised when multiple documents match the specified query.

```
User.create(name='Simon', email='simon@example.com')
User.create(name='Simon', email='simon@example.org')

try:
    user = User.get(name='Simon')
except User.MultipleDocumentsFound:
    """Handle the exception here"""
else:
    """Only one User was found"""
```

`NoDocumentFound` This exception is raised when no documents match the specified query.

```
try:
    user = User.get(name='Alvin')
except User.NoDocumentFound:
    """Handle the exception here"""
else:
    """Only one User was found"""
```

In the case of `NoDocumentFound`, there may be times when the way to handle the exception would be to create the document. A common pattern would:

```
try:
    user = User.get(name='Simon')
except User.NoDocumentFound:
    user = User.create(name='Simon')
```

Rather than making you use this pattern over and over, Simon does it for you, inside the `get_or_create()` method. Not only will `get_or_create()` do this, it will also let you know if it had to create the document.

```
user, created = User.get_or_create(name='Simon')
# user will be the newly created document and created will be True

user, created = User.get_or_create(name='Simon')
# user will be loaded from the database and created will be False
```

If multiple documents match the query, `MultipleDocumentsFound` will still be raised.

1.5 Saving

There are a couple of different approaches that can be taken when writing data to a MongoDB database. Simon provides a few different methods to perform writes to help expose the full power of each.

1.5.1 Document Replacement

The basic way to create or update a document is with the `save()` method. It will save the document associated with the instance to the database. If an update is being performed, the version of the document in the database will be overwritten by the version associated with the instance. This is known as document replacement. Any changes made to the version of the document in the database that have not been introduced to the instance will be lost.

```
user = User(name='Simon')
user.save()
```

This can be condensed into one step using the `create()` method.

```
user = User.create(name='Simon')
```

`save()` can also be used to save changes to a document.

```
user.email = 'simon@example.com'
user.save()
```

The first of these calls to `save()` will result in an `insert`. The second will result in an `update`. In the mongo Shell they would be written as:

```
db.users.insert({name: 'Simon'})

db.users.update({_id: ObjectId(...)}, {email: 'simon@example.com'})
```

1.5.2 Atomic Updates

MongoDB also offers a more powerful way to save changes to documents: atomic updates. By utilizing atomic updates, you can write selective changes to portions of a document without replacing the whole thing. Simon provides several different ways to perform atomic updates.

save_fields The `save_fields()` method will perform an atomic update updating only the specified fields.

```
# update only the score field
user.score = 100
user.save_fields('score')
```

You can also update multiple fields at once.

```
user.score = 200
user.friends = ['Alvin', 'Theodore']
user.save_fields(['score', 'friends'])
```

In the mongo Shell these would be:

```
db.users.update({_id: ObjectId(...)}, {$set: {score: 100}})

db.users.update({_id: ObjectId(...)}, {$set: {score: 200, friends: ['Alvin', 'Theodore']}})
```

update The `update()` method provides a shortcut to the behavior offered by `save_fields()`.

```
user.update(score=100)

user.update(score=200, friends=['Alvin', 'Theodore'])
```

increment The `increment()` method provides a way to increment the values of the specified fields. If the field does not exist, it will be added with the initial value of `0`.

When incrementing only one field, only the name of the field needs to be given to `increment()`. A value can also be provided if incrementing by any value other than `1`.

```
user.increment('score')

user.increment('score', 100)
```

`increment()` can also be used to increment multiple fields at once.

```
user.increment(score=100, level=1)
```

The equivalent queries in the mongo Shell would be:

```
db.users.update({_id: ObjectId(...)}, {$inc: {score: 1}})

db.users.update({_id: ObjectId(...)}, {$inc: {score: 100}})

db.users.update({_id: ObjectId(...)}, {$inc: {score: 100, level: 1}})
```

remove_fields The `remove_fields()` method will remove the specified fields from the document in the database.

Using it works just like `save_fields()`.

```
user.remove_fields('level')

user.remove_fields(['level', 'friends'])
```

To execute these same queries in the mongo Shell:

```
db.users.update({_id: ObjectId(...)}, {$unset: {level: 1}})

db.users.update({_id: ObjectId(...)}, {$unset: {level: 1, friends: 1}})
```

raw_update The `raw_update()` method allows any update query to be specified.

This method will let you execute any update that can't appropriately be expressed through one of the other methods. Just make sure you use it with caution as Simon can do little to protect you.

```
user.raw_update({'$set': {'level': 1}, '$inc': {'score': 100}, '$unset': {'friends': 1}})
```

This query would be passed through to MongoDB as:

```
db.users.update({_id: ObjectId(...)}, {$set: {level: 1}, $inc: {score: 100}, $unset: {friends: 1
```

1.5.3 Write Concern

When Simon was first started, the default behavior with MongoDB was to perform writes without write concern. This led to faster performance but had the potential for data loss. Queries performed with write concern enabled will request the result of `getLastError()` before returning execution to the application. More information is available in the MongoDB Docs.

Simon was built with respect for this behavior as the default. All of the methods discussed above as well as `delete()` accept an argument called `safe` that can override the default behavior.

```
user = User(name='Simon')
user.save(safe=True)

user.update(email='simon@example.com', safe=True)

user.delete(safe=True)
```

This also applies to the `get_or_create()` method discussed in *Querying*.

1.6 Connecting to a Database

As useful as Simon is, it's of no use without a database connection. Connecting to a database can be as simple as specifying its name.

```
from simon.connection import connect

connect(name='simon')
```

This will connect to the `mongod` instance running on `localhost` and use the database named `simon`.

Of course the `connect()` method can do more than just connect to a database. For starters, it can connect to another database.

```
connect(name='metrics')
```

This will use the same `mongod` instance as before, this time using the database named `metrics`.

When connecting to a database, each is given an alias. By default, Simon tries to use the name of the database as its alias. If all of your databases are part of the same MongoDB server, this won't be an issue. There may be times, however, when your databases are located in multiple locations. If two databases have the same name, the default alias behavior won't be sufficient. Fortunately you can assign any alias you want.

```
connect('localhost', name='simon')

connect('legacy-server', name='simon', alias='legacy-simon')
```

This will connect to `mongod` on `localhost` and use the `simon` database with the alias `simon`. It will also connect to `mongod` on `legacy-server` and use the `simon` database with the alias `legacy-simon`.

Before moving on to more advanced concepts, there's one more thing to point out. The first call to `connect()` will have two aliases, the name of the database and `default`. By default, all of your Simon models will use this connection.

1.6.1 Authentication

As a matter of best practice, it's a good idea to use authentication with your database. The `connect()` method accepts `username` and `password` parameters.

```
connect(name='simon', username='theuser', password='thepassword')
```

This will fail to connect to the database if the authentication fails.

1.6.2 Replica Sets

Another good idea when working with MongoDB is to use replica sets. `connect()` accepts a parameter named `replica_set` with the name of the replica set to use.

```
connect(name='simon', replica_set='simon-rs')
```

1.6.3 URI Connection String

The `connect()` method supports connecting to a database using a URI.

```
connect('mongodb://username:password@localhost:27017/simon?replicaSet=simon-rs')
```

Full details are available in the MongoDB Docs.

1.7 Model Meta Options

When defining a class, you can do more than just give it a name. By defining a class named `Meta` within a model, you can control several aspects of its behavior. Any options that are omitted will be given their default values, as show below.

```
class User(Model):
    class Meta:
        auto_timestamp = True
        collection = 'users'
        database = 'default'
        field_map = {'id': '_id'}
        map_id = True
        safe = True
        sort = None
```

1.7.1 auto_timestamp

By default, calling `save()` will cause the `created` and `modified` fields to update accordingly. Adding `auto_timestamp = False` to the `Meta` class will disable this behavior.

```
class Meta:
    auto_timestamp = False  # do not automatically add timestamps
```

1.7.2 collection

By default, the collection associated with a model will be the name of the model with an `s` appended to it. Adding `collection` to the `Meta` class will allow its value to altered.

```
class Meta:
    collection = 'simon'  # store documents in the simon collection
```

1.7.3 database

By default, all collections will be located in the default database. If you use the `connect()` method to connect to additional databases, the database to use with a model can be controlled by adding the `database` option to the `Meta` class.

```
connect('localhost', name='logs', alias='logs')

class Meta:
    database = 'logs'  # use the logs database
```

1.7.4 field_map and map_id

By default, the _id field of all models is exposed through the id attribute. Additional fields can be added to the mapping by including them in the field_map dictionary. The keys of the dictionary represent attribute names and the values represent the keys used in the document.

When the map_id option is True (the default), you can define a custom mapping without having to include 'id': '_id'. It will be added for you.

```
class Meta:
    # map friends to list_of_friends
    field_map = {'list_of_friends': 'friends'}
    map_id = False  # do not map _id to id
```

You can also use field_map to expose nested fields as top-level attributes.

```
class Meta:
    field_map = {'x': 'location.x', 'y': 'location.y'}
```

Why would you want to use this behavior? Unlike a relational database which stores its schema at the table level, MongoDB's dynamic schema requires key names to be stored as part of each document. The longer the names of your keys, the more storage space you will need (keep in mind this is only really a problem with extremely large collections). When using shortened key names, it may make the names harder to remember, resulting in code that is harder to read and maintain. By utilizing field_map, more meaningful names can be used in code while storing shorter variations in the database.

```
class User(Model):
    class Meta:
        field_map = {
            'first_name': 'fname',
            'last_name': 'lname',
            'location': 'loc',
        }

user = User.create(first_name='Simon', last_name='Seville',
                   location='Fresno, CA')
```

This query executing in the mongo Shell would look a little different:

```
db.users.insert({fname: 'Simon', lname: 'Seville', loc: 'Fresno, CA'})
```

1.7.5 required_fields

While Simon tries to expose MongoDB's dynamic schema by not enforcing a schema on a model, there may be times when you wish to make sure that a document contains certain fields before it is saved. You can designate a field as required by adding it to the required_fields option in the Meta class.

```
class Meta:
    required_fields = 'email'
```

With this setting, you wouldn't be able to save a document unless it contained an `email` field.

You can also require multiple fields.

```
class Meta:
    required_fields = ('email', 'name')
```

If you try to save a document that is missing andy of the required fields, `TypeError` will be raised.

1.7.6 safe

With the introduction of `MongoClient`, updates are performed with write concern enabled. Simon mimics this behavior by setting the `safe` option in the `Meta` class to `True`. To revert to the previous behavior seen in versions of PyMongo prior to 2.4, set the `safe` option to `False`. When write concern is disabled at the model level, it can still be used on a case by case basis by providing `safe=True` as a parameter to method calls.

```
class Meta:
    safe = False  # don't use write concern for this model by default
```

More information about write concern is available in the MongoDB Docs.

1.7.7 sort

By default, calls to `all()` and `find()` will use natural order for sorting. If you want to have a model default to a different sort order, you can do so by defining the `sort` option in the `Meta` class.

```
class Meta:
    sort = 'name'  # sort by name ascending
```

The default sort can also handle multiple fields.

```
class Meta:
    sort = ('name', 'email')  # sort by name and email ascending
```

For a explanation of how to take full advantage of the `sort` option, check out the `sort()` method.

More information about natural sort is available in the MongoDB Docs.

1.8 Simon API

The following is a look into the API inside Simon.

1.8.1 Connection

Manage database connections

simon.connection.**connect**(*host='localhost'*, *name=None*, *username=None*, *password=None*, *port=None*, *alias=None*, ***kwargs*)

Connects to a database.

Parameters

- **host** (*str.*) – Hostname, IP address, or MongoDB URI of the host.
- **name** – (optional) The name of the MongoDB database.

- **username** (*str.*) – (optional) The username to use for authentication.
- **password** (*str.*) – (optional) The password to use for authentication.
- **port** (*int.*) – (optional) The port of the MongoDB host.
- **alias** (*str.*) – (optional) An alias to use for accessing the database. If no value is provided, `name` will be used.
- ****kwargs** (***kwargs.*) – All other keyword arguments accepted by `pymongo.connection.Connection`.

Returns `pymongo.database.Database` – the database.

Raises `ConnectionError`

Changed in version 0.2.0: `connect()` now accepts `replica_set` as a kwarg, it is preferred over `replicaSet`New in version 0.1.0.

`simon.connection.`**`get_database`**(*name*)
Gets a reference to a database.

Parameters **name** (*str.*) – The name of the database.

Returns `pymongo.database.Database` – a database object.

New in version 0.1.0.

exception `simon.connection.`**`ConnectionError`**
Raised when a database connection cannot be opened. New in version 0.1.0.

1.8.2 Model

class `simon.`**`Model`**(***fields*)
The base class for all Simon models New in version 0.1.0.

classmethod **`all`**()
Returns all documents in the collection.

If `sort` has been defined on the `Meta` class it will be used to order the records. New in version 0.1.0.

classmethod **`create`**(*safe=False*, ***fields*)
Creates a new document and saves it to the database.

This is a convenience method to create a new document. It will instantiate a new `Model` from the keyword arguments, call `save()`, and return the instance.

If the model has the `required_fields` options set, a `TypeError` will be raised if any of the fields are not provided.

Parameters

- **safe** (*bool.*) – (optional) Whether to perform the create in safe mode.
- ****fields** (***kwargs.*) – Keyword arguments to add to the document.

Returns `Model` – the new document.

Raises `TypeError`

New in version 0.1.0.

`delete`(*safe=False*)
Deletes a single document from the database.

This will delete the document associated with the instance object. If the document does not have an `_id`– this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

Parameters **safe** (*bool.*) – (optional) Whether to perform the delete in safe mode.

Raises `TypeError`

New in version 0.1.0.

classmethod **`find`**(*q=None*, **qs*, ***fields*)

Gets multiple documents from the database.

This will find a return multiple documents matching the query specified through `**fields`. If `sort` has been defined on the `Meta` class it will be used to order the records.

Parameters

- **q** (`Q`.) – (optional) A logical query to use with the query.
- ***qs** (**args.*) – **DEPRECATED** Use `q` instead.
- ****fields** (***kwargs.*) – Keyword arguments specifying the query.

Returns `QuerySet` – query set containing objects matching `query`.

Changed in version 0.3.0: Deprecating `qs` in favor of `q`New in version 0.1.0.

classmethod **`get`**(*q=None*, **qs*, ***fields*)

Gets a single document from the database.

This will find and return a single document matching the query specified through `**fields`. An exception will be raised if any number of documents other than one is found.

Parameters

- **q** (`Q`.) – (optional) A logical query to use with the query.
- ***qs** (**args.*) – **DEPRECATED** Use `q` instead.
- ****fields** (***kwargs.*) – Keyword arguments specifying the query.

Returns `Model` – object matching `query`.

Raises `MultipleDocumentsFound`, `NoDocumentFound`

Changed in version 0.3.0: Deprecating `qs` in favor of `q`New in version 0.1.0.

classmethod **`get_or_create`**(*safe=False*, ***fields*)

Gets an existing or creates a new document.

This will find and return a single document matching the query specified through `**fields`. If no document is found, a new one will be created.

Along with returning the `Model` instance, a boolean value will also be returned to indicate whether or not the document was created.

Parameters

- **safe** (*bool.*) – (optional) Whether to perform the create in safe mode.
- ****fields** (***kwargs.*) – Keyword arguments specifying the query.

Returns tuple – the `Model` and whether the document was created.

Raises `MultipleDocumentsFound`

New in version 0.1.0.

increment (*field=None*, *value=1*, *safe=False*, ***fields*)
Performs an atomic increment.

This can be used to update a single field:

```
>>> obj.increment(field, value)
```

or to update multiple fields at a time:

```
>>> obj.increment(field1=value1, field2=value2)
```

Note that the latter does **not** set the values of the fields, but rather specifies the values they should be incremented by.

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

If no fields are indicated–either through `field` or through `**fields`, a `ValueError` will be raised.

Parameters

- **field** (*str.*) – (optional) Name of the field to increment.
- **value** (*int.*) – (optional) Value to increment `field` by.
- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.
- ****fields** (***kwargs.*) – Keyword arguments specifying fields and increment values.

Raises `TypeError`, `ValueError`

New in version 0.1.0.

pop (*fields*, *safe=False*)
Performs an atomic pop.

Values can be popped from either the end or the beginning of a list. To pop a value from the end of a list, specify the name of the field. The pop a value from the beginning of a list, specify the name of the field with a − in front of it.

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

Parameters

- **fields** (*str, list, or tuple.*) – The names of the fields to pop from.
- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.

Raises `TypeError`

New in version 0.5.0.

pull (*field=None*, *value=None*, *safe=False*, ***fields*)
Performs an atomic pull.

With MongoDB there are two types of pull operations: `$pull` and `$pullAll`. As the name implies, `$pullAll` is intended to pull all values in a list from the field, while `$pull` is meant for single values.

This method will determine the correct operator(s) to use based on the value(s) being pulled. Updates can consist of either operator alone or both together.

This can be used to update a single field:

```
>>> obj.pull(field, value)
```

or to update multiple fields at a time:

```
>>> obj.pull(field1=value1, field2=value2)
```

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

If no fields are indicated–either through `field` or through `**fields`, a `ValueError` will be raised.

Parameters

- **field** (*str.*) – (optional) Name of the field to pull from.
- **value** (*scalar or list.*) – (optional) Value to pull from `field`.
- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.
- ****fields** (***kwargs.*) – Keyword arguments specifying fields and the values to pull.

Raises `TypeError`, `ValueError`

New in version 0.5.0.

push (*field=None*, *value=None*, *allow_duplicates=True*, *safe=False*, ***fields*)
Performs an atomic push.

With MongoDB there are three types of push operations: `$push`, `$pushAll`, add `$addToSet`. As the name implies, `$pushAll` is intended to push all values from a list to the field, while `$push` is meant for single values. `$addToSet` can be used with either type of value, but it will only add a value to the list if it doesn't already contain the value.

This method will determine the correct operator(s) to use based on the value(s) being pushed. Setting `allow_duplicates` to `False` will use `$addToSet` instead of `$push` and `$pushAll`. Updates that allow duplicates can combine `$push` and `$pushAll` together.

This can be used to update a single field:

```
>>> obj.push(field, value)
```

or to update multiple fields at a time:

```
>>> obj.push(field1=value1, field2=value2)
```

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

If no fields are indicated–either through `field` or through `**fields`, a `ValueError` will be raised.

Parameters

- **field** (*str.*) – (optional) Name of the field to push to.
- **value** (*scalar or list.*) – (optional) Value to push to `field`.
- **allow_duplicates** (*bool.*) – (optional) Whether to allow duplicate values to be added to the list
- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.
- ****fields** (***kwargs.*) – Keyword arguments specifying fields and the values to push.

Raises `TypeError`, `ValueError`

New in version 0.5.0.

raw_update (*fields*, *safe=False*)
Performs an update using a raw document.

This method should be used carefully as it will perform the update exactly, potentially performing a full document replacement.

Also, for simple updates, it is preferred to use the `save()` or `update()` methods as they will usually result in less data being transferred back from the database.

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

Unlike `save()`, `modified` will not be updated.

Parameters

- **fields** (*dict.*) – The document to save to the database.
- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.

Raises `TypeError`

New in version 0.1.0.

remove_fields (*fields*, *safe=False*)
Removes the specified fields from the document.

The specified fields will be removed from the document in the database as well as the object. This operation cannot be undone.

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

Unlike `save()`, `modified` will not be updated.

If the model has the `required_fields` options set, a `TypeError` will be raised if attempting to remove one of the required fields.

Parameters

- **fields** (*str, list, or tuple.*) – The names of the fields to remove.
- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.

Raises `TypeError`

New in version 0.1.0.

rename (*field_from=None*, *field_to=None*, *safe=False*, ***fields*)
Performs an atomic rename.

This can be used to update a single field:

```
>>> obj.rename(original, new)
```

or to update multiple fields at a time:

```
>>> obj.increment(original1=new1, original2=new2)
```

Note that the latter does **not** set the values of the fields, but rather specifies the name they should be renamed to.

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

If no fields are indicated–either through `field_from` and `field_to` or through `**fields`, a `ValueError` will be raised.

Parameters

- **field_from** (*str.*) – (optional) Name of the field to rename.
- **field_to** (*int.*) – (optional) New name for `field_from`.
- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.
- ****fields** (***kwargs.*) – Keyword arguments specifying fields and their new names.

Raises `TypeError`, `ValueError`

New in version 0.5.0.

save (*safe=False*)
Saves the object to the database.

When saving a new document for a model with `auto_timestamp` set to `True`, `created` will be added with the current datetime in UTC. `modified` will always be set with the current datetime in UTC.

If the model has the `required_fields` options set, a `TypeError` will be raised if any of the fields have not been associated with the instance.

Parameters **safe** (*bool.*) – (optional) Whether to perform the save in safe mode.

Raises `TypeError`

Changed in version 0.4.0: `created` is always added to inserted documents when `auto_timestamp` is `True`New in version 0.1.0.

save_fields (*fields*, *safe=False*)
Saves only the specified fields.

If only a select number of fields need to be updated, an atomic update is preferred over a document replacement. `save_fields()` takes either a single field name or a list of field names to update.

All of the specified fields must exist or an `AttributeError` will be raised. To add a field to the document with a blank value, make sure to assign it through `object.attribute = ''` or something similar before calling `save_fields()`.

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

Unlike `save()`, `modified` will not be updated.

Parameters

- **fields** (*str, list, or tuple.*) – The names of the fields to update.
- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.

Raises `AttributeError`, `TypeError`

New in version 0.1.0.

update (*safe=False*, ***fields*)
Performs an atomic update.

If only a select number of fields need to be updated, an atomic update is preferred over a document replacement. `update()` takes a series of fields and values through its keyword arguments. This fields will be updated both in the database and on the instance.

If the document does not have an `_id`–this will most likely indicate that the document has never been saved– a `TypeError` will be raised.

Unlike `save()`, `modified` will not be updated.

Parameters

- **safe** (*bool.*) – (optional) Whether to perform the update in safe mode.
- ****fields** (***kwargs.*) – The fields to update.

Raises `TypeError`

New in version 0.1.0.

1.8.3 Geo

Helper methods to ease geospatial queries

`simon.geo.box`(*lower_left_point*, *upper_right_point*)

Builds a `$box` query.

This is a convenience method for `$within` queries that use `$box` as their shape.

`lower_left_point` and `upper_right_point` are a pair of coordinates, each as a `list`, that combine to define the bounds of the box in which to search.

Parameters

- **lower_left_point** (*list.*) – The lower-left bound of the box.
- **upper_right_point** (*list.*) – The upper-right bound of the box.

Returns dict – the `$box` query.

Raises `TypeError`, `ValueError`.

New in version 0.1.0.

`simon.geo.circle`(*point*, *radius*)

Builds a `$circle` query.

This is a convenience method for `$within` queries that use `$circle` as their shape.

Parameters

- **point** (*list.*) – The center of the circle.
- **radius** (*int.*) – The distance from the center of the circle.

Returns dict – the `$circle` query.

Raises `TypeError`, `ValueError`.

New in version 0.1.0.

`simon.geo.near`(*point*, *max_distance=None*, *unique_docs=False*)

Builds a `$near` query.

This is a convenience method for more complex `$near` queries. For simple queries that simply use the point, the regular query syntax of `field__near=[x, y]` will suffice. This method provides a way to include `$maxDistance` and (if support is added) `$uniqueDocs` without needing to structure the query as `field={'$near': [x, y], '$maxDistance': z}`.

Note 2012-11-29 As of the current release of MongoDB (2.2), `$near` queries do not support the `$uniqueDocs` parameter. It is included here so that when support is added to MongoDB, no changes to the library will be needed.

Parameters

- **point** (*list, containing exactly two elements.*) – The point to use for the geospatial lookup.

- **max_distance** (*int.*) – (optional) The maximum distance a point can be from `point`.
- **unique_docs** – (optional) If `True` will only return unique documents.

Returns dict – the `$near` query.

Raises `TypeError`, `ValueError`.

New in version 0.1.0.

`simon.geo.`**`polygon`**`(*points)`
Builds a `$polygon` query.

This is a convenience method for `$within` queries that use `$polygon` as their shape.

`points` should either be expressed as a series of `list`'s or a single `dict` containing `dict`'s providing pairs of coordinates that behind the polygon.

Parameters ***points** (**args.*) – The bounds of the polygon.

Returns dict – the `$polygon` query.

Raises `TypeError`, `ValueError`.

New in version 0.1.0.

`simon.geo.`**`within`**`(shape, *bounds, **bounds_map)`
Builds a `$within` query.

This is a convenience method for `$within` queries.

Parameters

- **shape** (*str.*) – The shape of the bounding area.
- ***bounds** (**args.*) – Coordinate pairs defining the bounding area.
- ****bounds_map** (***kwargs.*) – Named coordinate pairs defining the bounding area.

Returns dict – the `$within` query.

Raises `RuntimeError`.

New in version 0.1.0.

1.8.4 Query

Query functionality

class `simon.query.`**`Q`**`(**fields)`
A wrapper around a query condition to allow for logical ANDs and ORs through `&` and `|`, respectively. New in version 0.1.0.

class `simon.query.`**`QuerySet`**`(cursor=None, cls=None)`
A query set that wraps around MongoDB cursors and returns `Model` objects. New in version 0.1.0.

`count``()`
Gets the number of documents in the `QuerySet`.

If no cursor has been associated with the query set, `TypeError` will be raised.

Returns int – the number of documents.

Raises `TypeError`.

New in version 0.1.0.

distinct(*key*)
Gets a list of distinct values for `key` across all documents in the `QuerySet`.

Parameters **key** (*str.*) – Name of the key.

Returns list – distinct values for the key.

New in version 0.1.0.

limit(*limit*)
Applies a limit to the number of documents in the `QuerySet`.

Parameters **limit** (*int.*) – Number of documents to return.

Returns `QuerySet` – the documents with the limit applied.

New in version 0.1.0.

skip(*skip*)
Skips a number of documents in the `QuerySet`.

Parameters **skip** (*int.*) – Number of documents to skip.

Returns `QuerySet` – the documents remaining.

New in version 0.1.0.

sort(**keys*)
Sorts the documents in the `QuerySet`.

By default all sorting is done in ascending order. To switch any key to sort in descending order, place a - before the name of the key.

```
>>> qs.sort('id')
>>> qs.sort('grade', '-score')
```

Parameters ***keys** (**args.*) – Names of the fields to sort by.

Returns `QuerySet` – the sorted documents.

New in version 0.1.0.

1.8.5 Utils

Helper utilities

WARNING The functionality in this module is intended for internal use by Simon. If using anything in this module directly, be careful when updating versions of Simon as no guarantees are made about the backward compatability of its API.

simon.utils.**current_datetime**()
Gets the current datetime in UTC formatted for MongoDB

Python includes microseconds in its `datetime` values. MongoDB, on the other hand, only retains them down to milliseconds. This method will not only get the current time in UTC, but it will also remove microseconds from the value.

Returns datetime – the current datetime formatted for MongoDB.

New in version 0.2.0.

simon.utils.**get_nested_key**(*values*, *key*)
Gets a value for a nested dictionary key.

This method can be used to retrieve the value nested within a dictionary. The entire path should be provided as the value for `key`, using a `.` as the delimiter (e.g., `'path.to.the.key'`).

If `key` does not exist in `values`, `KeyError` will be raised. The exception will be raised in the reverse order of the recursion so that the original value is used.

Parameters

- **values** (*dict.*) – The dictionary.
- **key'** – The path of the nested key.

Returns The value associated with the nested key.

Raises `KeyError`

New in version 0.1.0.

simon.utils.**guarantee_object_id**(*value*)
Converts a value into an Object ID.

This method will convert a value to an `ObjectId`. If `value` is a `dict` (e.g., with a comparison operator as the key), the value in the `dict` will be converted. Any values that are a `list` or `tuple` will be iterated over, and replaced with a `list` containing all `ObjectId` instances.

`TypeError` will be raised for any `value` that cannot be converted to an `ObjectId`. `InvalidId` will be raised for any `value` that is of the right type but is not a valid value for an `ObjectId`.

Any value of `None` will be replaced with a newly generated `ObjectId`.

Parameters **value** – the ID.

Returns ObjectId or dict – the Object ID.

Raises `TypeError`, `InvalidId`

New in version 0.1.0.

simon.utils.**is_atomic**(*document*)
Checks for atomic update operators.

This method checks for operators in `document`. If a spec document is provided instead of a document to save or update, a false positive will be reported if a logical operator such as `$and` or `$or` is used.

Parameters **document** (*dict.*) – The document to containing the update.

Returns bool – `True` if `document` is an atomic update.

New in version 0.3.0.

simon.utils.**map_fields**(*cls*, *fields*, *with_operators=False*, *flatten_keys=False*)
Maps attribute names to document keys.

Attribute names will be mapped to document keys using `cls._meta.field_map`. If any of the attribute names contain `__`, `parse_kwargs()` will be called and a second pass through `cls._meta.field_map` will be performed.

The two-pass approach is used to allow for keys in embedded documents to be mapped. Without the first pass, only keys of the root document could be mapped. Without the second pass, only keys that do not contain embedded document could be mapped.

The $and and $or operators cannot be mapped to different keys. Any occurrences of these operators as keys should be accompanied by a list of dict``s. Each ``dict will be put back into map_fields() to ensure that keys nested within boolean queries are mapped properly.

If with_operators is set, the following operators will be checked for and included in the result:

- $gt the key's value is greater than the value given
- $gte the key's value is greater than or equal to the value given
- $lt the key's value is less than the value given
- $lte the key's value is less than or equal to the value given
- $ne the key's value is not equal to the value given
- $all the key's value matches all values in the given list
- $in the key's value matches a value in the given list
- $nin the key's value is not within the given list
- $exists the the key exists
- $near the key's value is near the given location
- $size the key's value has a length equal to the given value
- $elemMatch the key's value is an array satisfying the given query

To utilize any of the operators, append __ and the name of the operator sans the $ (e.g., __gt, __lt) to the name of the key:

```
map_fields(ModelClass, {'a__gt': 1, 'b__lt': 2},
           with_operators=True)
```

This will check for a greater than 1 and b less than 2 as:

```
{'a': {'$gt': 1}, 'b': {'$lt': 2}}
```

The $not operator can be used in conjunction with any of the above operators:

```
map_fields(ModelClass, {'a__gt': 1, 'b__not__lt': 2},
           with_operators=True)
```

This will check for a greater than 1 and b not less than 2 as:

```
{'a': {'$gt': 1}, 'b': {'$not': {'$lt': 2}}}
```

If flatten_keys is set, all keys will be kept at the top level of the result dictionary, using a . to separate each part of a key. When this happens, the second pass will be omitted.

Parameters

- **cls** (*type.*) – A subclass of Model.
- **fields** (*dict.*) – Key/value pairs to be used for queries.
- **with_operators** (*bool.*) – (optional) Whether or not to process operators.
- **flatten_keys** (*bool.*) – (optional) Whether to allow the nested keys to be nested.

Returns dict – key/value pairs renamed based on cls's field_map mapping.

New in version 0.1.0.

simon.utils.**parse_kwargs**(***kwargs*)
Parses embedded documents from dictionary keys.

This takes a kwargs dictionary whose keys contain ___ and convert them to a new dictionary with new keys created by splitting the originals on the ___.

Parameters ****kwargs** (***kwargs.*) – Keyword arguments to parse.

Returns dict – dictionary with nested keys generated from the names of the arguments.

New in version 0.1.0.

simon.utils.**remove_nested_key**(*original*, *key*)
Removes keys within a nested dictionary.

This method can remove a key from within a nested dictionary. Nested keys should be specified using a `.` as the delimiter. If no delimiter is found, the key will be removed from the root dictionary.

If `original` is not a dictionary, a `TypeError` will be raised. If `key` doesn't exist in `original`, a `KeyError` will be raised.

Parameters

- **original** (*dict.*) – The original dictionary to be updated.
- **key** (*str.*) – The key to be removed.

Returns dict – the updated dictionary

Raises `TypeError`, `KeyError`

New in version 0.1.0.

simon.utils.**update_nested_keys**(*original*, *updates*)
Updates keys within nested dictionaries.

This method simulates merging two dictionaries. It allows specific keys within a dictionary or nested dictionary without overwriting the the entire dictionary.

If either `original` or `updates` is not a dictionary, a `TypeError` will be raised.

Parameters

- **original** (*dict.*) – The original dictionary to be updated.
- **updates** (*dict.*) – The dictionary with updates to apply.

Returns dict – the updated dictionary.

Raises `TypeError`

New in version 0.1.0.

CHAPTER

TWO

INSTALLATION

To install the latest stable version of Simon:

```
$ pip install Simon
```

or, if you must:

```
$ easy_install Simon
```

To install the latest development version:

```
$ git clone git@github.com:dirn/Simon.git
$ cd Simon
$ python setup.py install
```

CHAPTER

THREE

FURTHER READING

For more information, check out the PyMongo docs and the MongoDB docs.

CHAPTER

FOUR

CHANGELOG

4.1 0.5.0 (2013-02-21)

- Add `pop()` method

‘ Add `pull()` method - Add `push()` method - Add `rename()` method - Add support for `$elemMatch` operator

4.2 0.4.0 (2013-02-12)

- `created` will be set for all inserted documents whose model has `auto_timestamp` set to `True`
- Fix `create()` bug

4.3 0.3.0 (2013-02-11)

- Deprecate `Model.get()` and `Model.find()` argument `qs` in favor of `q`
- Correctly specify write concern depending on version of PyMongo
- Refactor database interaction
- Bug fixes

4.4 0.2.0 (2013-02-03)

- Change `connection.connect()` argument from `replicaSet` to `replica_set`
- Add equality comparisons for models
- Add support for required fields
- Use write concern by default

4.5 0.1.0 (2013-01-18)

- Initial release

CHAPTER

FIVE

INDICES AND TABLES

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

PYTHON MODULE INDEX

S