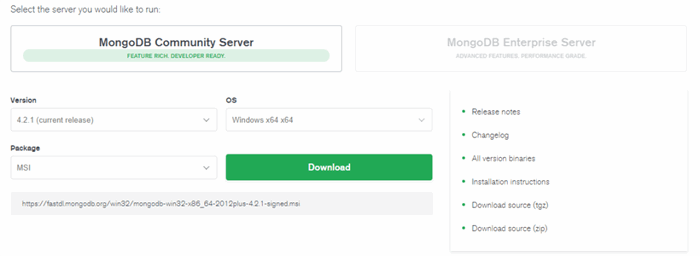
# Using MongoDB in Python

In this article, we implement one **repository**, using which you can manipulate data in the collection *Users*. We present **two ways** to achive that, ie. using two *Python* modules – *PyMongo*and *MongoEngine*. The implementation with first library *PyMongo* is**low-level** implementation, while the implementation with *MongoEngine* can be observed as **higher-level** implementation. However, before we get into nitty-gritty details of each implementation, let’s first **install***MongoDB* and both *Python* modules.

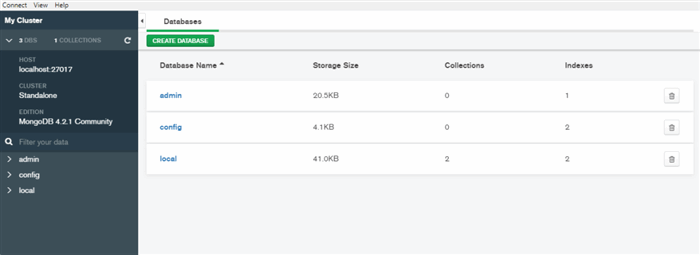
Prerequisites

*MongoDB*is free and **open-source**. It can be downloaded from [**here**](https://www.mongodb.com/download-center?jmp=nav#community). After picking up version and your **operating system**. In this example, we use *MongoDB 4.2.1 Comunity Server* for *Windows*.

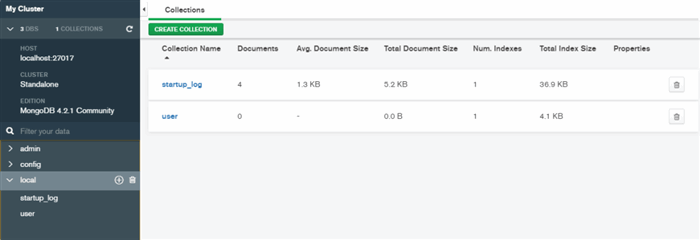
After installation, a user can run *MongoDB*server by using the **command**:

mongod –dbpath PATH\_TO\_THE\_DIR

With this installation, *MongoCompass* is installed as well. That is visual **GUI**component, using which you can manipulate your databases, collections and documents:



For the purpose of these examples, we use *local* database and *user* collection:



As we mentioned you are going to need two *Python* modules, so let’s install them. First, you can install *PyMongo* like this:

pip install pymongo

After that *MongoEngine*:

pip install mongoengine

And that is pretty much it. We are ready for implementations.

PyMongo Implementation

Let’s implement *UserRepository* class is using *PyMongo*module. To be more precise, we need just one **class**from this module – *MongoClient*:

from pymongo import MongoClient

[In the constructor of the *UserRepository*, we create an **instance**of *MongoClient* and connect to the cluster. There are several ways you can achieve that, and you can see two ways in the code. Then you pick database and collection. The rest of the implementation is more less just **wrapping**functions from *self.\_user* field.](https://gist.github.com/NMZivkovic/277335f96b707a82a00d183916382e66/raw/15bf9c185c466fed6c2e73a5b905adc33029768f/import_pymongo.py)

class UserRepository(object):

def \_\_init\_\_(self):

mongoclient = MongoClient('localhost', 27017)

*#mongoclient = MongoClient('mongodb://localhost:27017')*

database = mongoclient.local

self.\_users = database.user

Now, we are able to **create**an object of our class like this:

user\_repo = UserRepository()

Create Operations

Next, we **implement***create* operations. We add two methods: *insert* and *insert\_many*. First method adds just **one***user* to the collection, while the other adds all the users from the provided **array**. We can see that underneath they are just utilizing functions *insert\_one* and *insert\_many* from *PyMongo*.

class UserRepository(object):

def \_\_init\_\_(self):

mongoclient = MongoClient('localhost', 27017)

*#mongoclient = MongoClient('mongodb://localhost:27017')*

database = mongoclient.local

self.\_users = database.user

*# Create operations*

def insert(self, user):

return self.\_users.insert\_one(user)

def insert\_many(self, users):

return self.\_users.insert\_many(users)

Now, we can use these **functions**from our object. Method *insert* can be used like this:

user = {

'name':"Rubik",

'age':33,

'blog':"rubikscode.net"

}

result = user\_repo.insert(user)

print(result.acknowledged)

When we **peek**into this collection using *MongoDB Compass* we can see that the document is created in *user* collection.

While *insert\_many* can be used like this:

user\_1 = {

'name':"Vanja",

'age':29,

'blog':"eventroom.org"

}

user\_2 = {

'name':"Marko",

'age':36,

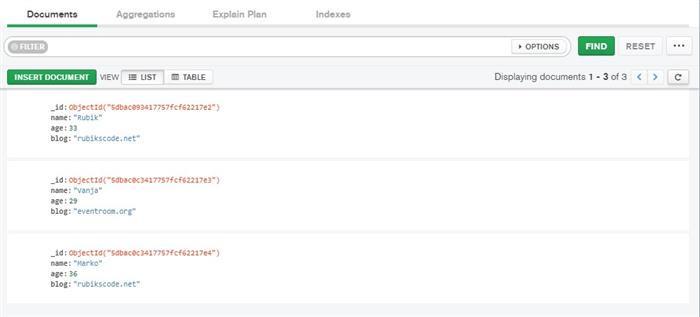
'blog':"rubikscode.net"

}

result = user\_repo.insert\_many([user\_1, user\_2])

print(result.acknowledged)

The result of this operation once again can be checked **visually**using *MongoDB Compass*:



Read Operations

Cool, now when we can add documents into our collection, let’s implement functions that are able to **retrieve**this information. We add three new functions: *read\_all*, *read\_many* and *read*. Function *read\_all* returns **all**documents from the collection. Methods *read* and *read\_many* are similar, they both accept some kind of **conditions**. The first one, however, returns just one document that **satisfies**the conditions, while others return all documents that satisfy the condition.

class UserRepository(object):

def \_\_init\_\_(self):

mongoclient = MongoClient('localhost', 27017)

*#mongoclient = MongoClient('mongodb://localhost:27017')*

database = mongoclient.local

self.\_users = database.user

*# Create operations*

def insert(self, user):

return self.\_users.insert\_one(user)

def insert\_many(self, users):

return self.\_users.insert\_many(users)

*# Read operations*

def read\_all(self):

return self.\_users.find()

def read\_many(self, conditions):

return self.\_users.find(conditions)

def read(self, conditions):

return self.\_users.find\_one(conditions)

These functions can be **used**like this:

print("---Read All---")

all\_users = user\_repo.read\_all()

for user in all\_users:

print(user)

print("--------------\n")

print("---Read Many--")

rc\_users = user\_repo.read\_many({'blog':'rubikscode.net'})

for user in rc\_users:

print(user)

print("--------------\n")

print("--- Read ---")

one\_rc\_user = user\_repo.read({'blog':'rubikscode.net'})

print(one\_rc\_user)

print("--------------\n")

And the **output**looks like this:

—Read All—

{‘\_id’: ObjectId(‘5dbad5a4dde338d6e1cd3ea7’), ‘name’: ‘Rubik’, ‘age’: 33, ‘blog’: ‘rubikscode.net’}

{‘\_id’: ObjectId(‘5dbc2f46ac3310f0215ba277’), ‘name’: ‘Vanja’, ‘age’: 29, ‘blog’: ‘eventroom.org’}

{‘\_id’: ObjectId(‘5dbc2f46ac3310f0215ba278’), ‘name’: ‘Marko’, ‘age’: 36, ‘blog’: ‘rubikscode.net’}

————–

—Read Many–

{‘\_id’: ObjectId(‘5dbad5a4dde338d6e1cd3ea7’), ‘name’: ‘Rubik’, ‘age’: 33, ‘blog’: ‘rubikscode.net’}

{‘\_id’: ObjectId(‘5dbc2f46ac3310f0215ba278’), ‘name’: ‘Marko’, ‘age’: 36, ‘blog’: ‘rubikscode.net’}

————–

— Read —

{‘\_id’: ObjectId(‘5dbad5a4dde338d6e1cd3ea7’), ‘name’: ‘Rubik’, ‘age’: 33, ‘blog’: ‘rubikscode.net’}

————–

Update Operations

We implement two update methods: *update* and *increase\_age*. The purpose of the first one is to demonstrate how you can update **single**document using conditions. The other function demonstrates how you can **encapsulate***MongoDB* update **modifiers**within functions and tailor repository for your needs. This is arguably the biggest **advantage**of this approach.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

class UserRepository(object):

def \_\_init\_\_(self):

mongoclient = MongoClient('localhost', 27017)

*#mongoclient = MongoClient('mongodb://localhost:27017')*

database = mongoclient.local

self.\_users = database.user

*# Create operations*

def insert(self, user):

return self.\_users.insert\_one(user)

def insert\_many(self, users):

return self.\_users.insert\_many(users)

*# Read operations*

def read\_all(self):

return self.\_users.find()

def read\_many(self, conditions):

return self.\_users.find(conditions)

def read(self, conditions):

return self.\_users.find\_one(conditions)

*# Update operations*

def update(self, conditions, new\_value):

return self.\_users.update\_one(conditions, new\_value)

def increment\_age(self, conditions):

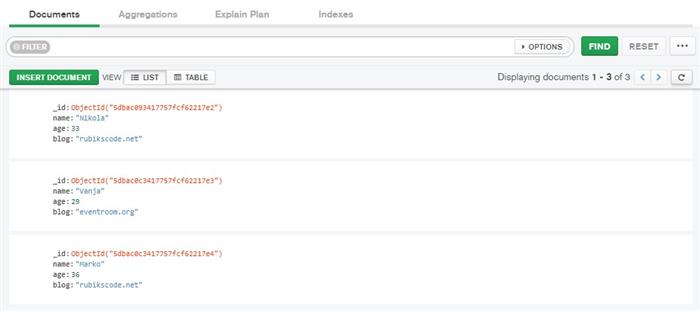
return self.\_users.update\_one(conditions, {'$inc' : {'age' : 1}})

Function *update* can be used to update *name* of a document like this:

result = user\_repo.update({'name':'Rubik'}, {'$set': {'name': "Nikola"}})

print(result.acknowledged)

The **result**is this:

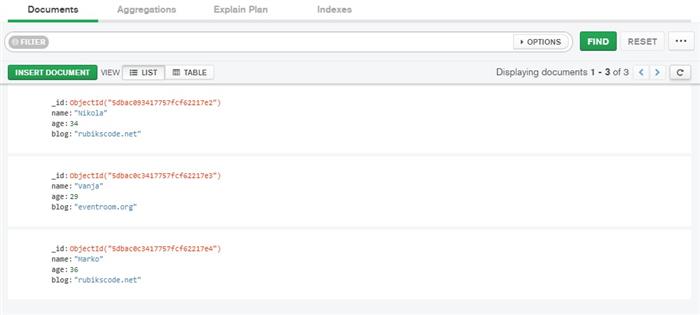


Method *increment\_age* is even easier to **use**:

result = user\_repo.increment\_age({'name':'Nikola'})

print(result.acknowledged)

**Result**can be observed in *MongoDB Compass*:



Delete Operations

Just like we could create one or multiple documents at once we can delete one or multiple documents. This is done using two **functions**: *delete* and *delete\_many*. Both of these functions receive conditions for deletion. Here is what the **final form** of *UserRepository* class looks like:

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

class UserRepository(object):

def \_\_init\_\_(self):

mongoclient = MongoClient('localhost', 27017)

*#mongoclient = MongoClient('mongodb://localhost:27017')*

database = mongoclient.local

self.\_users = database.user

*# Create operations*

def insert(self, user):

return self.\_users.insert\_one(user)

def insert\_many(self, users):

return self.\_users.insert\_many(users)

*# Read operations*

def read\_all(self):

return self.\_users.find()

def read\_many(self, conditions):

return self.\_users.find(conditions)

def read(self, conditions):

return self.\_users.find\_one(conditions)

*# Update operations*

def update(self, conditions, new\_value):

return self.\_users.update\_one(conditions, new\_value)

def increment\_age(self, conditions):

return self.\_users.update\_one(conditions, {'$inc' : {'age' : 1}})

*# Delete operations*

def delete(self, condition):

return self.\_users.delete\_one(condition)

def delete\_many(self, condition):

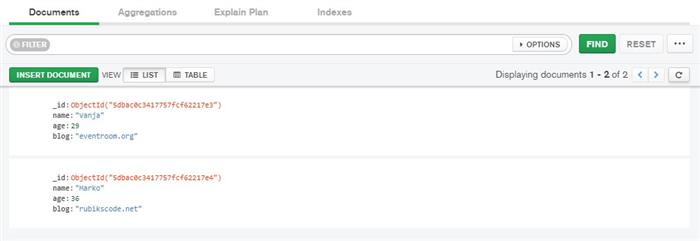
return self.\_users.delete\_many(condition)

If we want to delete **one**document, we can do it like this:

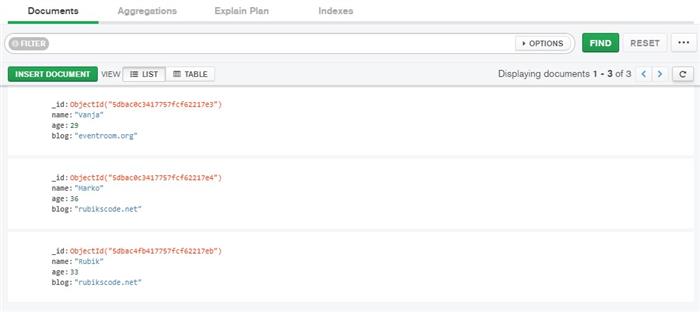
result = user\_repo.delete({'name':'Nikola'})

print(result.acknowledged)

The **result**of this operation:



Before we demonstrate what deletion of multiple documents look like, we add a document that we just deleted and restore collection in the **previous**state:

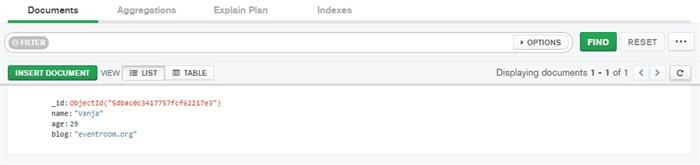


If after this we call *delete\_many* that would look like this:

result = user\_repo.delete\_many({'blog':'rubikscode.net'})

print(result.acknowledged)

**Result** of this operation:



MongoEngine Implementation

As you were able to see, implementing repository using *PyMongo* is quite easy and fun. However, many users don’t need to steep to this **low-level** of abstractions. That is why some people opt to use **library**that is built on top of *PyMongo* – *MongoEngine*. In essence, *MongoEngine* is **ODM**, ie. Object Document Mapper. Just like we have **ORMs**for relational databases, for document NoSQL databases we have **ODM**s. *MongoEngine* provides required **higher-level** of abstraction. So, let’s import it and attach to our database:

from mongoengine import \*

connect('local', host='localhost', port=27017)

The first parameter of the function is the **name**of the database, while the other two are defining **location**. In order to “*attach*” to collection, we need to implement a **class**that describes documents from that collection – **model**. This is very similar to the way we do it in various *ORMs*, so if you have experience with, let’s say, *EntityFramework* this will come natural to you. The model class has to **inherit***Document.* For our *user* collection that looks like this:

class User(Document):

name = StringField(required=True, max\_length=50)

age = IntField(required=True)

blog = StringField(required=True, max\_length=50)

In this model, we’ve told *MongoEngine*that we expect *User* to have *name, age* and *blog.* Underneath, *Document* base object **validates**data that is provided. *MongoEngine* provides various field **types**(here we used just *StringField* and *IntField*) and **options**for those types.

Create Operation

When we implemented *User* model class, it is quite easy to **manipulate**the database objects with it. Here is how we can create document in *user* collection:

user = User(

name = "Rubik",

age = 33,

blog = "rubikscode.net")

*# Create Operation*

user.save()

Note that *save* method **creates**document in *user* collection in this particular case.



Read Operations

Reading data is also **simplified**. Here is how it is done:

read\_users = User.objects

for user in User.objects:

print("Name: {}; Age: {}; Blog: {}".format(user.name, user.age, user.blog))

Using *objects* field we can access all documents in the database. In this case there is only one:

Name: Rubik; Age: 33; Blog: rubikscode.net

We can utilize *filter* function on top of that if we want to create various **queries**.

Update Operation

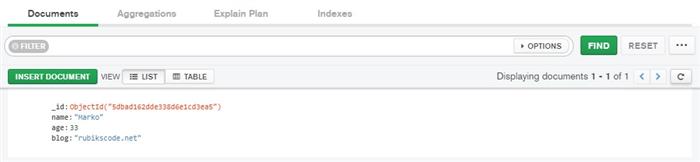
Update is done again using *save* method on **existing**object. Basically, when you call *save* method for the first time, it will create a document, while the second time it will **update**it.

user.name = "Marko"

*# Update*

user.save()

The result:

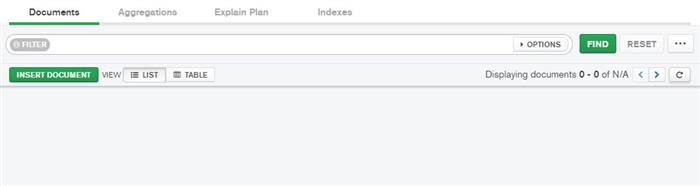


Delete Operation

Delete operation is performed simply by calling *delete* method on *User* object.

user.delete()

The **result**looks like this:



Conclusion

In this article, we see how we can use *MongoDB* in *Python*. We could see that they are pretty **comparable**and function well together. We used two modules *PyMongo* and *MongoEngine*, so depending on the level of abstraction you need, you can pick the one that **suits**your project the most.

Thank you for reading!