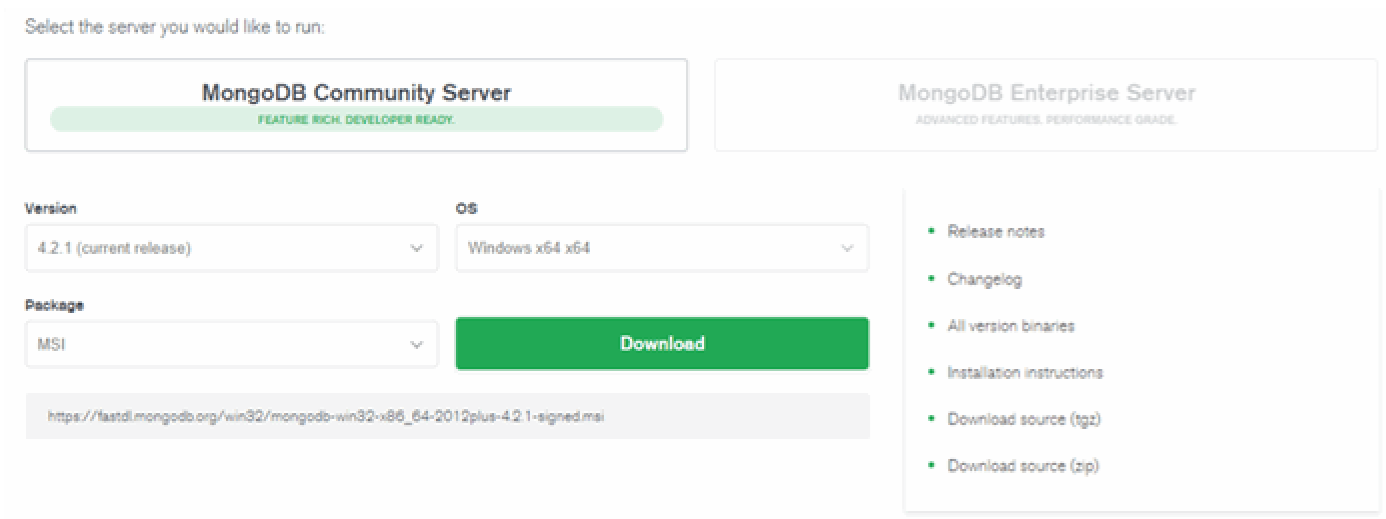
Using MongoDB in Python

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**. After picking up version and your **operating system**. In this example, we use *MongoDB 4.2.1 Comunity Server* for *Windows*.


Select the server you would like to run:
MongoDB Community Server
FEATURE RICH. DEVELOPER READY.
MongoDB Enterprise Server
ADVANCED FEATURES. PERFORMANCE GRADE.
Download
Release notes
Changelog
All version binaries
Installation instructions
Download source (tgz)
Download source (zip)

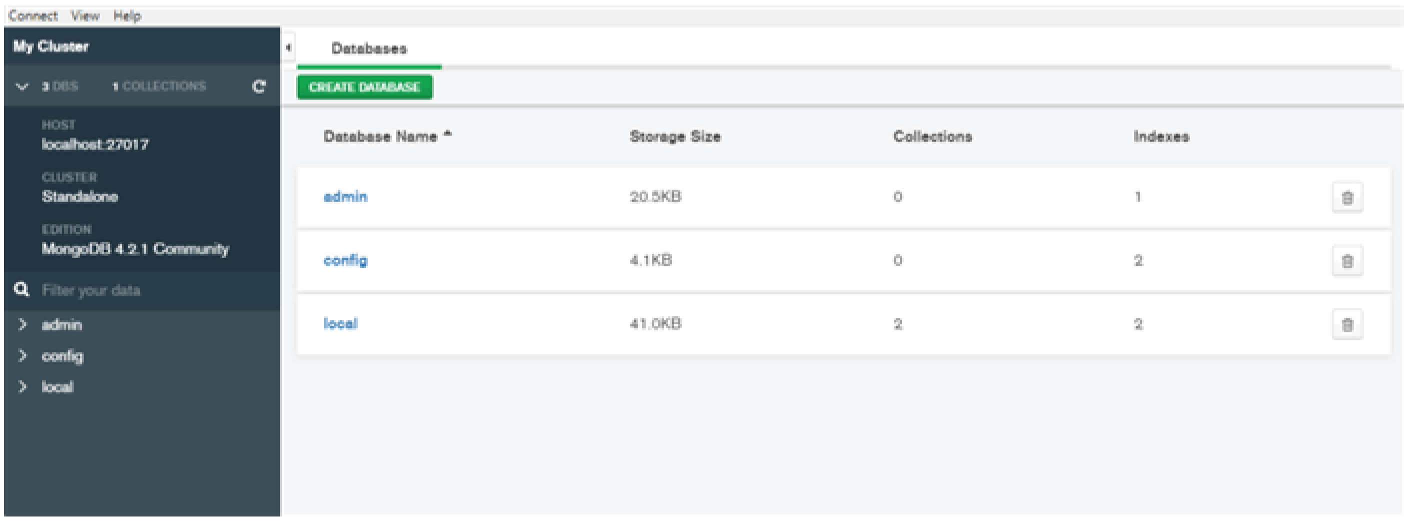
Version: 4.2.1 (current release)
OS: Windows x64 x64
Package: MSI

```
https://fastdl.mongodb.org/win32/mongodb-win32-x86_64-2012plus-4.2.1-signed.msi
```

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

```
mongod -dbpath PATH_TO_THE_DIR
```

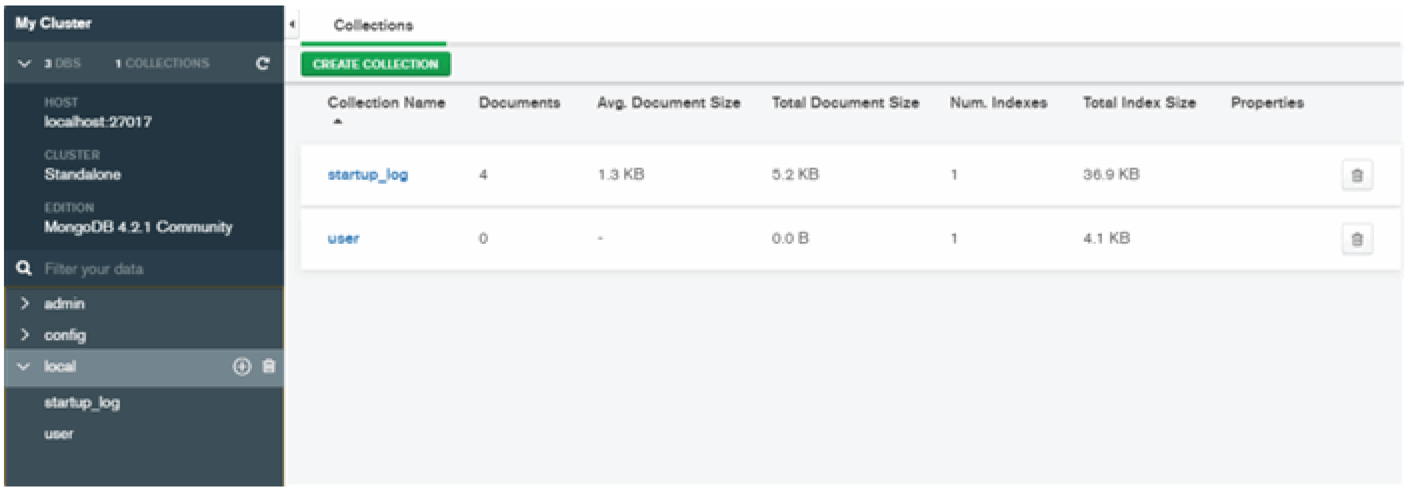
With this installation, *MongoCompass* is installed as well. That is visual **GUI** component, using which you can manipulate your databases, collections and documents:



HOST
localhost:27017
CLUSTER
Standalone
EDITION
MongoDB 4.2.1 Community

Database Name	Storage Size	Collections	Indexes
admin	20.5KB	0	1
config	4.1KB	0	2
local	41.0KB	2	2

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



HOST
localhost:27017
CLUSTER
Standalone
EDITION
MongoDB 4.2.1 Community

Collection Name	Documents	Avg. Document Size	Total Document Size	Num. Indexes	Total Index Size	Properties
startup_log	4	1.3 KB	5.2 KB	1	36.9 KB	
user	0	-	0.0 B	1	4.1 KB	

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.

```
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*.

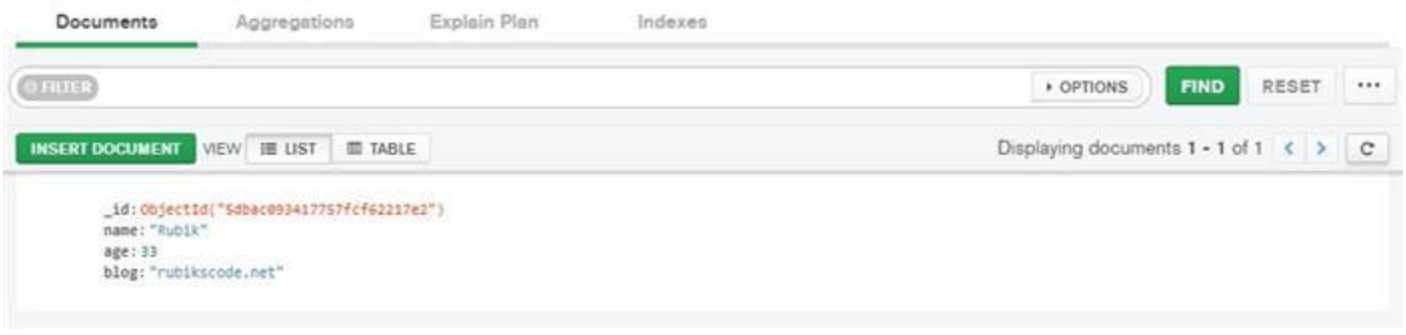
```
class UserRepository(object):
    def __init__(self):
        mongoclient = MongoClient('localhost', 27017)
        #mongoclient = MongoClient('mongodb://localhost:27017')

        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.

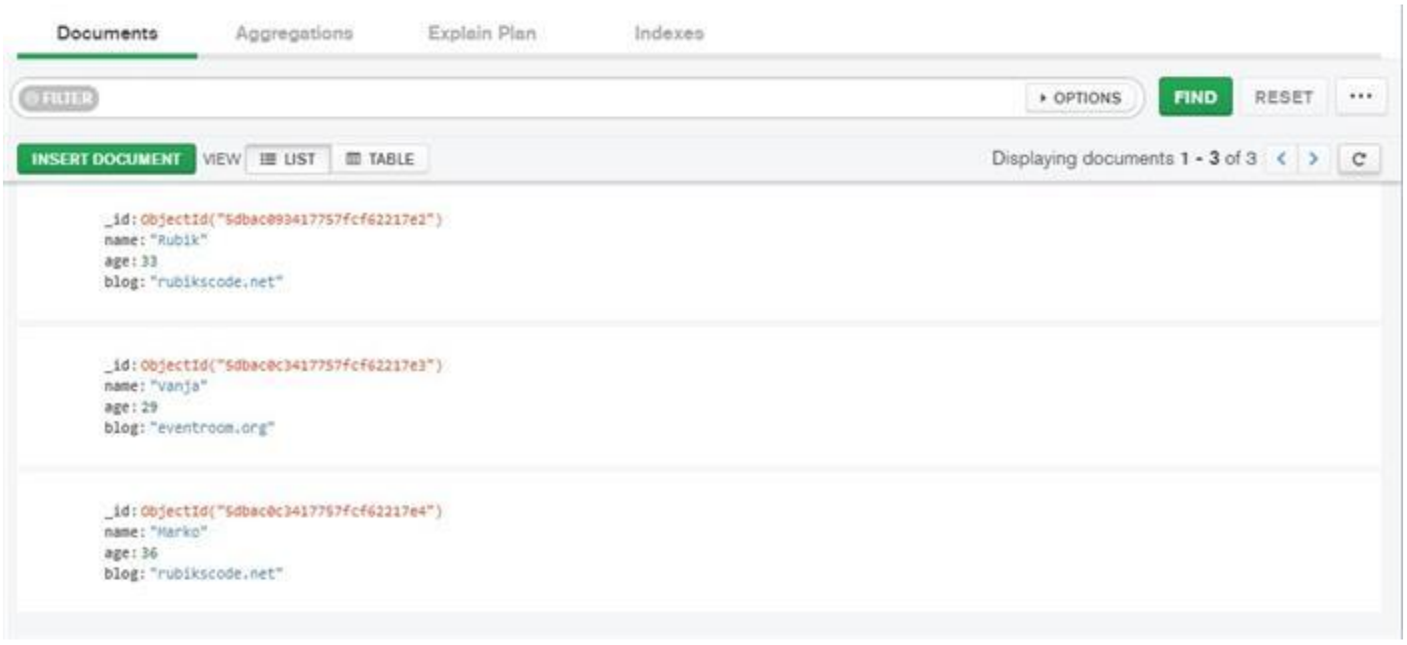


```
_id: ObjectId("5dbac093417757fcf62217e2")
name: "Rubik"
age: 33
blog: "rubikscode.net"
```

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*:



```
_id: ObjectId("5dbac093417757fcf62217e2")
name: "Rubik"
age: 33
blog: "rubikscode.net"

_id: ObjectId("5dbac0c3417757fcf62217e3")
name: "Vanja"
age: 29
blog: "eventroom.org"

_id: ObjectId("5dbac0c3417757fcf62217e4")
name: "Marko"
age: 36
blog: "rubikscode.net"
```

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 Copy Code

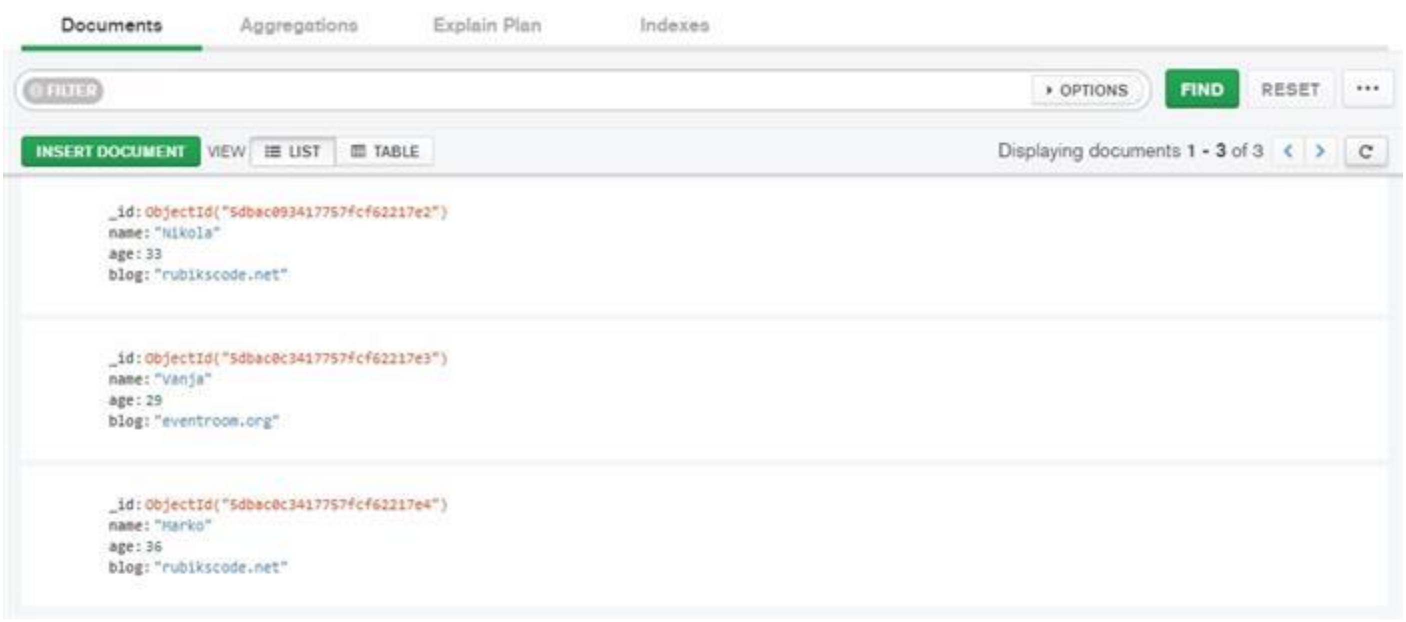
```python
class UserRepository(object):
    def __init__(self):
        mongoclient = MongoClient('localhost', 27017)
        #mongoclient = MongoClient('mongodb://localhost:27017')

        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:

```python
result = user_repo.update({'name':'Rubik'}, {'$set': {'name': "Nikola"}})
```

```python
print(result.acknowledged)
```

The **result** is this:

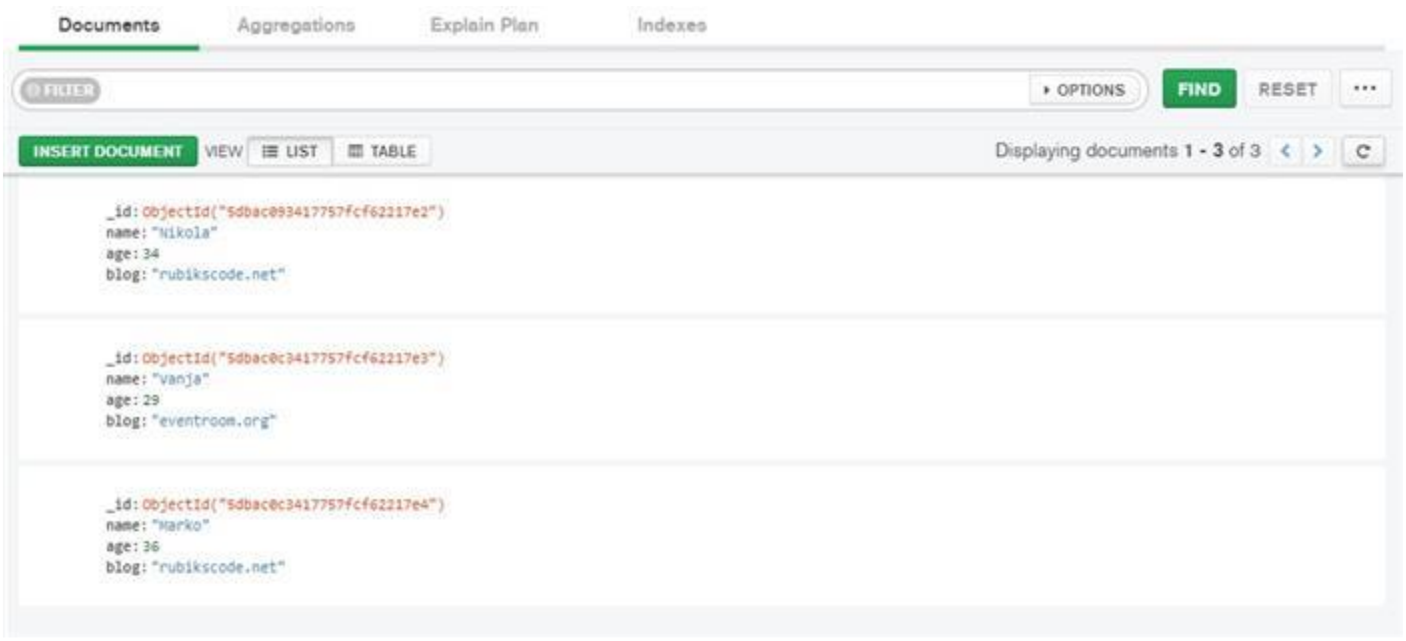


```
_id: ObjectId("5dbac093417757fcf62217e2")
name: "Nikola"
age: 33
blog: "rubikscode.net"

_id: ObjectId("5dbac0c3417757fcf62217e3")
name: "Vanja"
age: 29
blog: "eventroom.org"

_id: ObjectId("5dbac0c3417757fcf62217e4")
name: "Marko"
age: 36
blog: "rubikscode.net"
```

Method *increment_age* is even easier to **use**:

```python
result = user_repo.increment_age({'name':'Nikola'})
print(result.acknowledged)
```

Result can be observed in *MongoDB Compass*:



```
_id: ObjectId("5dbac093417757fcf62217e2")
name: "Nikola"
age: 34
blog: "rubikscode.net"

_id: ObjectId("5dbac0c3417757fcf62217e3")
name: "Vanja"
age: 29
blog: "eventroom.org"

_id: ObjectId("5dbac0c3417757fcf62217e4")
name: "Marko"
age: 36
blog: "rubikscode.net"
```

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:

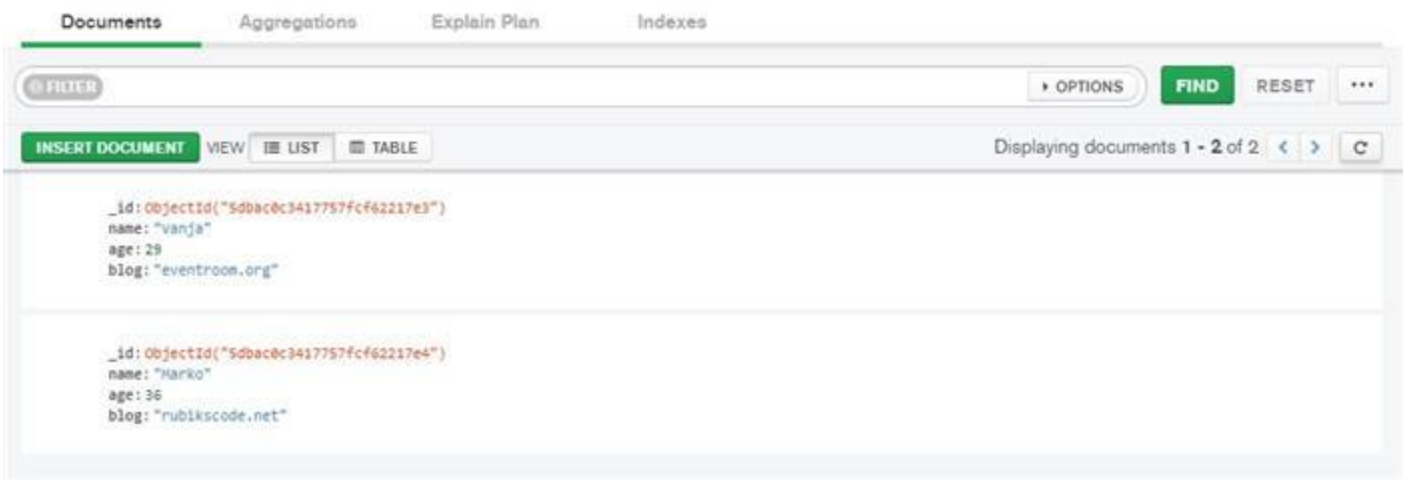
Hide Shrink ▲ Copy Code

```
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:

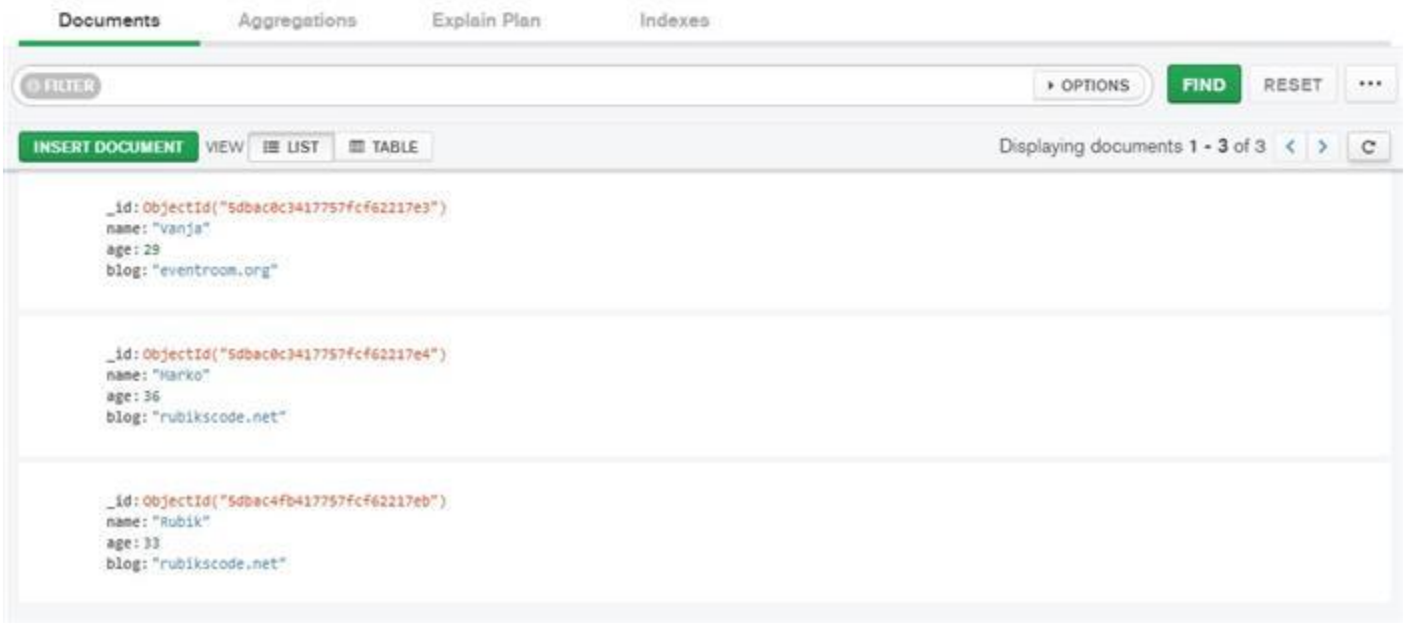
```
result = user_repo.delete({'name':'Nikola'})
print(result.acknowledged)
```

The **result** of this operation:



```
_id: ObjectId("5dbac0c3417757fcf62217e3")
name: "Vanja"
age: 29
blog: "eventroom.org"

_id: ObjectId("5dbac0c3417757fcf62217e4")
name: "Marko"
age: 36
blog: "rubikscode.net"
```

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:

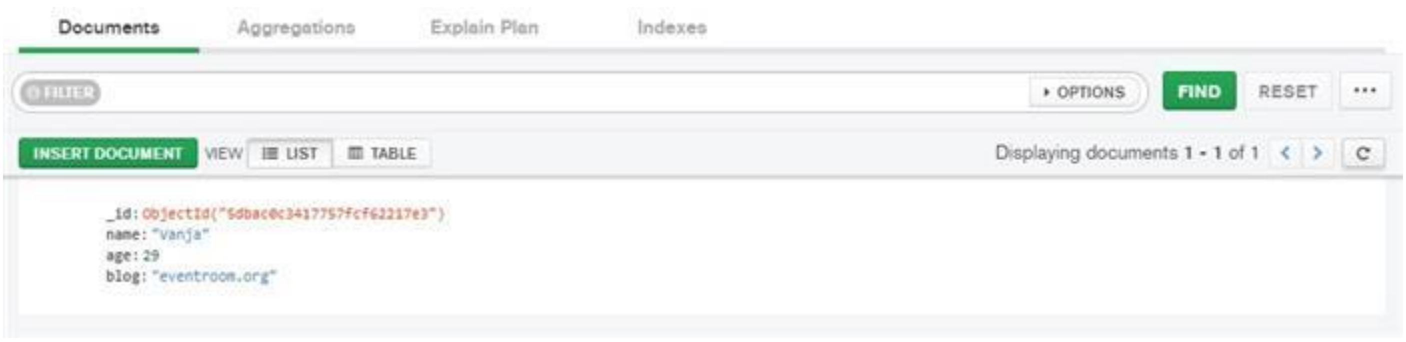


```
_id: ObjectId("5dbac0c3417757fcf62217e3")
name: "Vanja"
age: 29
blog: "eventroom.org"

_id: ObjectId("5dbac0c3417757fcf62217e4")
name: "Marko"
age: 36
blog: "rubikscode.net"

_id: ObjectId("5dbac4fb417757fcf62217eb")
name: "Rubik"
age: 33
blog: "rubikscode.net"
```

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:



```
_id: ObjectId("5dbac0c3417757fcf62217e3")
name: "Vanja"
age: 29
blog: "eventroom.org"
```

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)
```

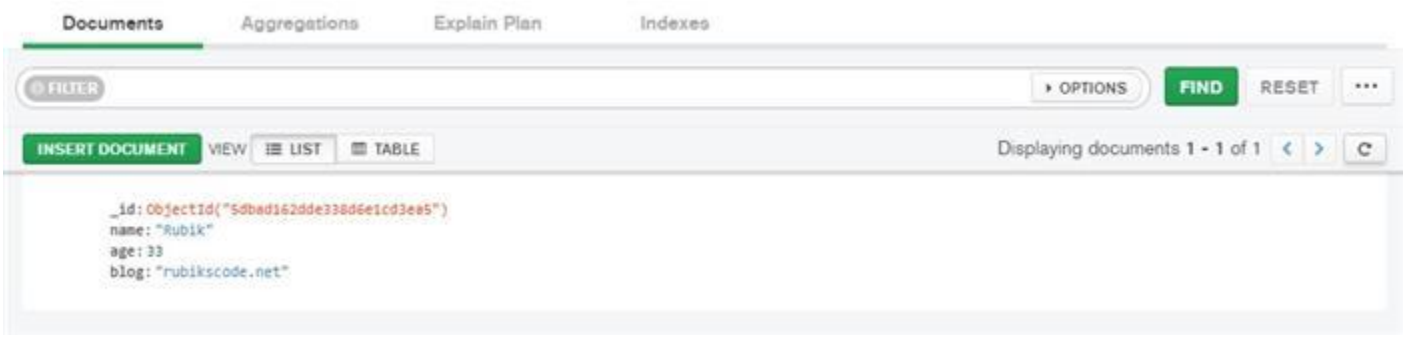
In this model, we've told *MongoEngine* that we expect *User* to have *name, age* and *blog*. Underneath, *Document* base object **validates** data that is provided. *MongoEngine* provides various field **types** (here we used just *StringField* and *IntField*) and **options** for those types.

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.



```
_id: ObjectId("5dbad162dde338d6e1cd3ea5")
name: "Rubik"
age: 33
blog: "rubikscode.net"
```

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
```

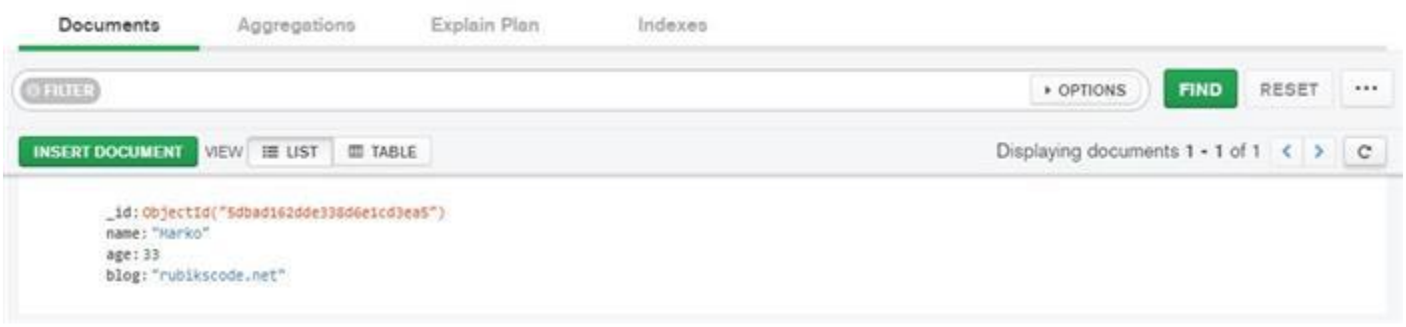
We can utilize *filter* function on top of that if we want to create various **queries**.

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:



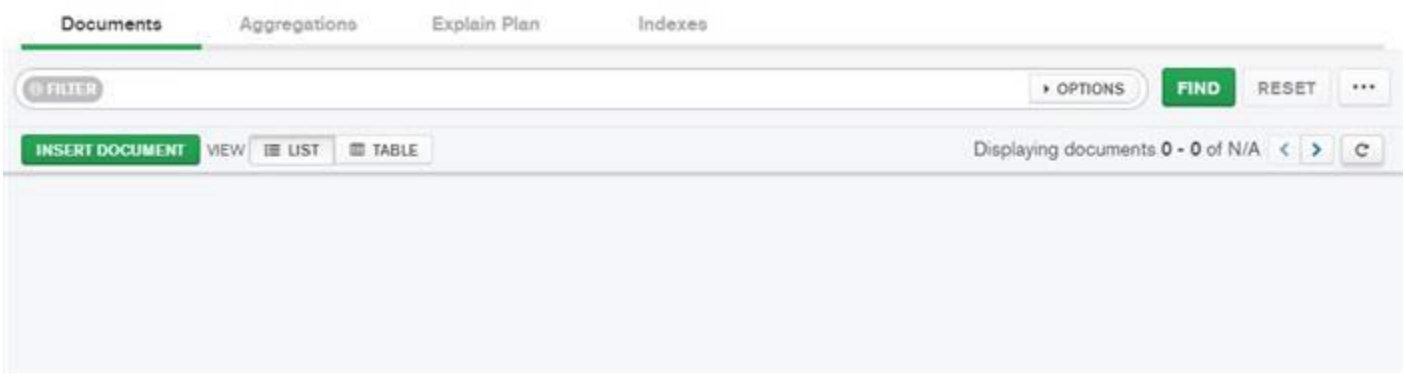
```
_id: ObjectId("5dbad162dde338d6e1cd3ea5")
name: "Marko"
age: 33
blog: "rubikscode.net"
```

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!