Install MongoDB

On your virtual machine, first update the APT package repository cache with the following command:

```
sudo apt update
```

Now install *wget* and *gnupg*

```
sudo apt install wget gnupg
```

Add the GPG key of the MongoDB official package repository to your Debian installation.

```
wget -qO - https://www.mongodb.org/static/pgp/server-4.2.asc | sudo apt-key add -
```

Add the official MongoDB 4 package repository for Debian 9 on Debian 10

```
echo "deb http://repo.mongodb.org/apt/debian buster/mongodb-org/4.2 main" | \
sudo tee /etc/apt/sources.list.d/mongodb-org-4.2.list
```

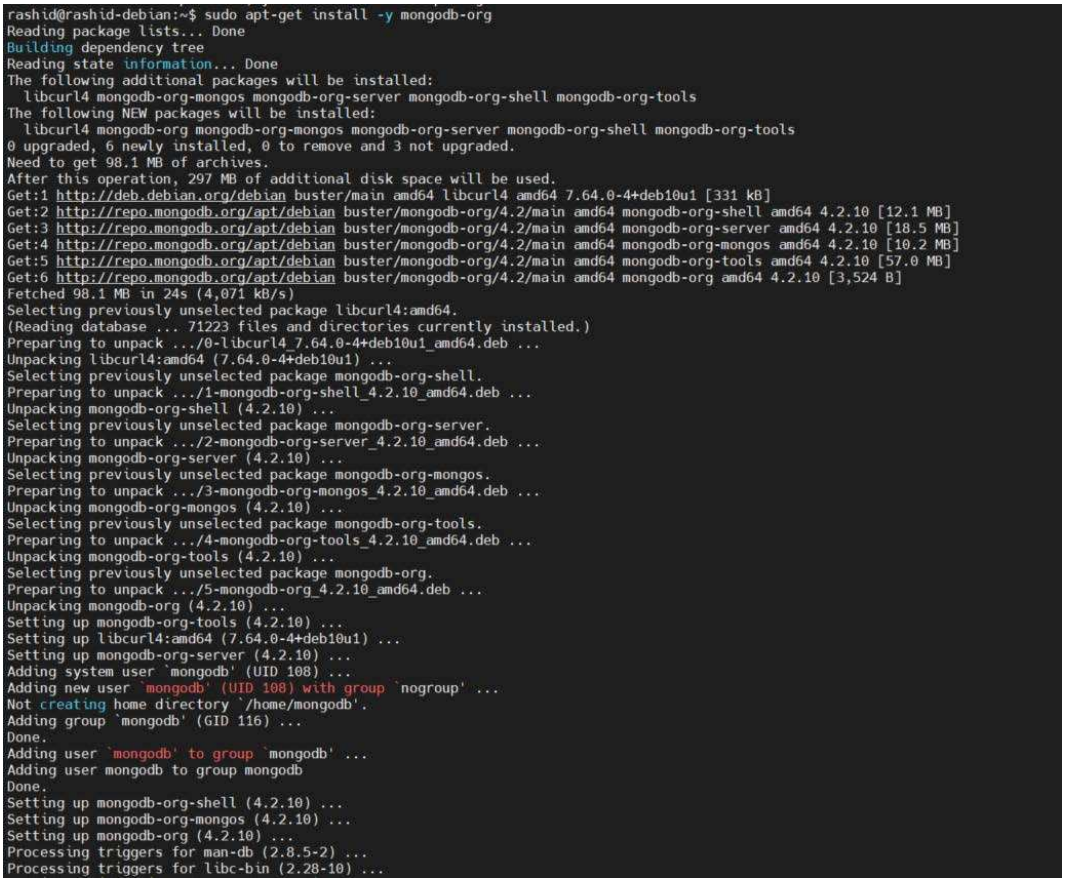
```
echo "deb http://deb.debian.org/debian/ stretch main" | \
sudo tee /etc/apt/sources.list.d/debian-stretch.list
```

Install MongoDB 4.

```
sudo apt update
sudo apt-get install -y mongodb-org
```

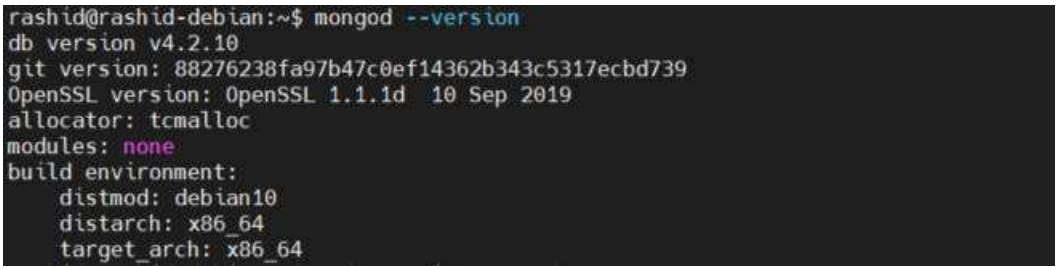
Enable **MongoDB**, which will cause it to automatically start when the virtual machine is booted.

sudo systemctl enable **mongod**



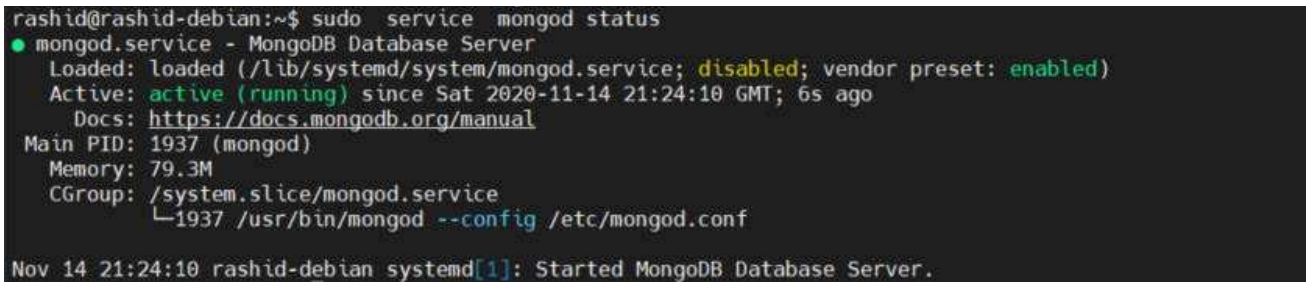
Now check whether MongoDB 4 is working correctly.

```
mongod --version
```



Finally, start the MongoDB service and check its status.

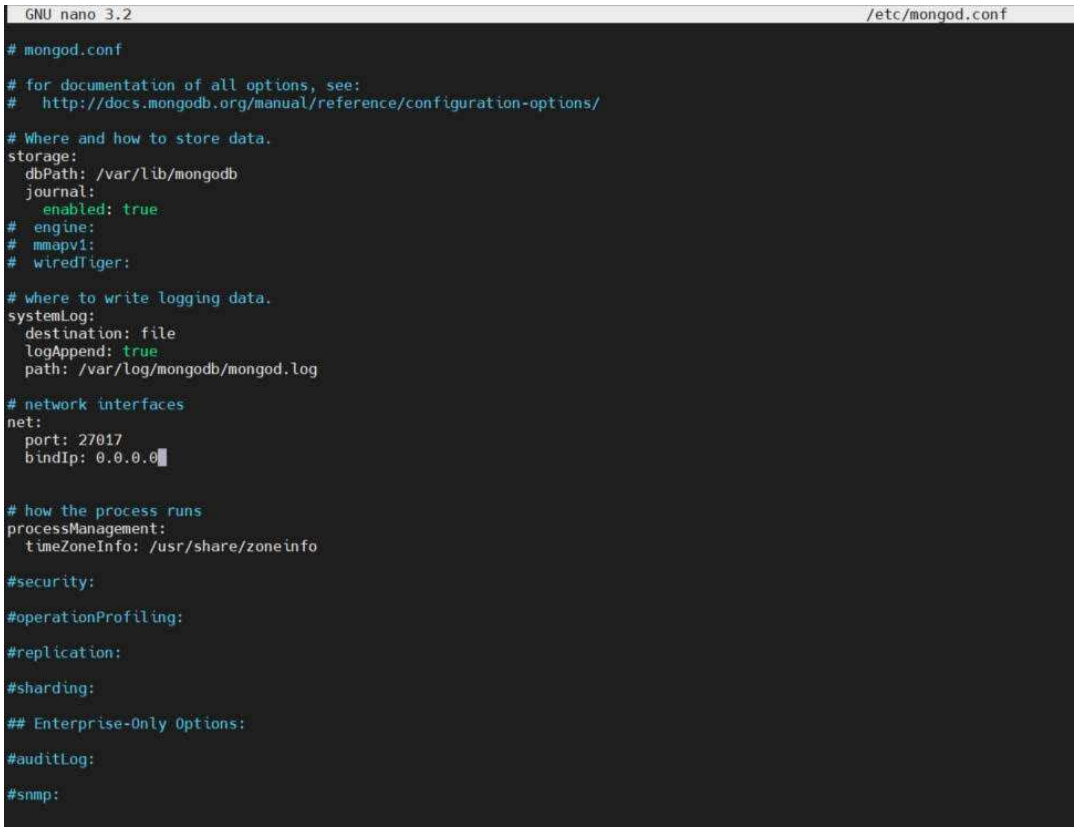
```
sudo service mongod start
sudo  service  mongod status
```



Accessing your Database remotely

Edit the /etc/mongod.conf file and set your bind_ip = 0.0.0.0 in order to be able to connect from outside the virtual machine.

sudo nano /etc/mongod.conf



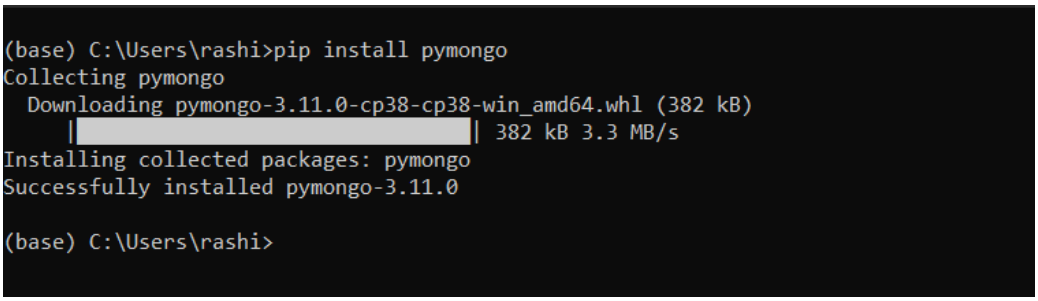
Restart MongoDB: **sudo service mongod restart**

Install pymongo

We will now install pymongo – which contains tools for interacting with MongoDB database from Python

Open the **anaconda terminal on your main/host computer** and run the following command:

pip install pymongo



PyMongo

November 14, 2020

https://api.mongodb.com/python/current/tutorial.html

1 Making a Connection with MongoClient

[3]:
```
from pymongo import MongoClient
client = MongoClient('192.168.56.30', 27017)
```

2 Getting a Database

[13]:
```
db = client.test_database
```

3 Getting a Collection

[5]:
```
collection = db.test_collection
```

4 Documents

[6]:
```
import datetime
post = {"author": "Mike",
    "text": "My first blog post!",
    "tags": ["mongodb", "python", "pymongo"],
    "date": datetime.datetime.utcnow()}
```

5 Inserting a Document

[7]:
```
posts = db.posts
post_id = posts.insert_one(post).inserted_id
post_id
```

[7]:
```
ObjectId('5fb04fc934aa675cb92f1370')
```

6 listing all of the collections in our database

[8]:
```
db.list_collection_names()
```

[8]:
```
['posts']
```

7 Getting a Single Document With find_one()

The most basic type of query that can be performed in MongoDB is find_one(). This method returns a single document matching a query (or None if there are no matches). It is useful when you know there is only one matching document, or are only interested in the first match. Here we use find_one() to get the first document from the posts collection:

[11]:
```
import pprint
pprint.pprint(posts.find_one())
```

```
{'_id': ObjectId('5fb04fc934aa675cb92f1370'),
 'author': 'Mike',
 'date': datetime.datetime(2020, 11, 14, 21, 44, 17, 843000),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
```

find_one() also supports querying on specific elements that the resulting document must match. To limit our results to a document with author "Mike" we do:

[10]:
```
pprint.pprint(posts.find_one({"author": "Mike"}))
```

```
{'_id': ObjectId('5fb04fc934aa675cb92f1370'),
 'author': 'Mike',
 'date': datetime.datetime(2020, 11, 14, 21, 44, 17, 843000),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
```

If we try with a different author, like "Eliot", we'll get no result

[12]:
```
posts.find_one({"author": "Eliot"})
```

8 Querying By ObjectId

We can also find a post by its _id, which in our example is an ObjectId:

[14]:
```
post_id
```

[14]:
```
ObjectId('5fb04fc934aa675cb92f1370')
```

[15]:
```
pprint.pprint(posts.find_one({"_id": post_id}))
```

```
{'_id': ObjectId('5fb04fc934aa675cb92f1370'),
 'author': 'Mike',
 'date': datetime.datetime(2020, 11, 14, 21, 44, 17, 843000),
```

```
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
```

9 Bulk Inserts

In order to make querying a little more interesting, let's insert a few more documents. In addition to inserting a single document, we can also perform bulk insert operations, by passing a list as the first argument to insert_many(). This will insert each document in the list, sending only a single command to the server:

[17]:
```
new_posts = [{"author": "Mike",
              "text": "Another post!",
              "tags": ["bulk", "insert"],
              "date": datetime.datetime(2009, 11, 12, 11, 14)},
             {"author": "Eliot",
              "title": "MongoDB is fun",
              "text": "and pretty easy too!",
              "date": datetime.datetime(2009, 11, 10, 10, 45)}]
result = posts.insert_many(new_posts)
result.inserted_ids
```

[17]:
```
[ObjectId('5fb0515034aa675cb92f1373'), ObjectId('5fb0515034aa675cb92f1374')]
```

There are a couple of interesting things to note about this example:

The result from insert_many() now returns two ObjectId instances, one for each inserted document. new_posts[1] has a different "shape" than the other posts - there is no "tags" field and we've added a new field, "title". This is what we mean when we say that MongoDB is schema-free

10 Querying for More Than One Document

To get more than a single document as the result of a query we use the find() method. find() returns a Cursor instance, which allows us to iterate over all matching documents. For example, we can iterate over every document in the posts collection:

[18]:
```
all_posts = posts.find()
for post in all_posts:
    pprint.pprint(post)
```

```
{'_id': ObjectId('5fb04fc934aa675cb92f1370'),
 'author': 'Mike',
 'date': datetime.datetime(2020, 11, 14, 21, 44, 17, 843000),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
{'_id': ObjectId('5fb0514434aa675cb92f1371'),
 'author': 'Mike',
 'date': datetime.datetime(2009, 11, 12, 11, 14),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
```

```
{'_id': ObjectId('5fb0514434aa675cb92f1372'),
 'author': 'Eliot',
 'date': datetime.datetime(2009, 11, 10, 10, 45),
 'text': 'and pretty easy too!',
 'title': 'MongoDB is fun'}
{'_id': ObjectId('5fb0515034aa675cb92f1373'),
 'author': 'Mike',
 'date': datetime.datetime(2009, 11, 12, 11, 14),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
{'_id': ObjectId('5fb0515034aa675cb92f1374'),
 'author': 'Eliot',
 'date': datetime.datetime(2009, 11, 10, 10, 45),
 'text': 'and pretty easy too!',
 'title': 'MongoDB is fun'}
```

Just like we did with find_one(), we can pass a document to find() to limit the returned results. Here, we get only those documents whose author is "Mike":

[19]:
```
for post in posts.find({"author": "Mike"}):
    pprint.pprint(post)
```

```
{'_id': ObjectId('5fb04fc934aa675cb92f1370'),
 'author': 'Mike',
 'date': datetime.datetime(2020, 11, 14, 21, 44, 17, 843000),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
{'_id': ObjectId('5fb0514434aa675cb92f1371'),
 'author': 'Mike',
 'date': datetime.datetime(2009, 11, 12, 11, 14),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
{'_id': ObjectId('5fb0515034aa675cb92f1373'),
 'author': 'Mike',
 'date': datetime.datetime(2009, 11, 12, 11, 14),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
```

11 Counting

If we just want to know how many documents match a query we can perform a count_documents() operation instead of a full query. We can get a count of all of the documents in a collection:

[20]:
```
posts.count_documents({})
```

[20]: 5

or just of those documents that match a specific query:

[21]:
```
posts.count_documents({"author": "Mike"})
```

[21]:
```
3
```

12 Range Queries

MongoDB supports many different types of advanced queries. As an example, lets perform a query where we limit results to posts older than a certain date, but also sort the results by author. Here we use the special "$lt" operator to do a range query, and also call sort() to sort the results by author.

[22]:
```
d = datetime.datetime(2009, 11, 12, 12)
for post in posts.find({"date": {"$lt": d}}).sort("author"):
    pprint.pprint(post)
```

```
{'_id': ObjectId('5fb0514434aa675cb92f1372'),
 'author': 'Eliot',
 'date': datetime.datetime(2009, 11, 10, 10, 45),
 'text': 'and pretty easy too!',
 'title': 'MongoDB is fun'}
{'_id': ObjectId('5fb0515034aa675cb92f1374'),
 'author': 'Eliot',
 'date': datetime.datetime(2009, 11, 10, 10, 45),
 'text': 'and pretty easy too!',
 'title': 'MongoDB is fun'}
{'_id': ObjectId('5fb0514434aa675cb92f1371'),
 'author': 'Mike',
 'date': datetime.datetime(2009, 11, 12, 11, 14),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
{'_id': ObjectId('5fb0515034aa675cb92f1373'),
 'author': 'Mike',
 'date': datetime.datetime(2009, 11, 12, 11, 14),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
```

13 Indexing

Adding indexes can help accelerate certain queries and can also add additional functionality to querying and storing documents. In this example, we'll demonstrate how to create a unique index on a key that rejects documents whose value for that key already exists in the index.

First, we'll need to create the index:

[23]:
```
result = db.profiles.create_index([('user_id', pymongo.ASCENDING)],unique=True)
sorted(list(db.profiles.index_information()))
```

[23]:
```
['_id_', 'user_id_1']
```

Notice that we have two indexes now: one is the index on _id that MongoDB creates automatically, and the other is the index on user_id we just created.

Now let's set up some user profiles:

[24]:
```
user_profiles = [{'user_id': 211, 'name': 'Luke'},{'user_id': 212, 'name':␣
↪'Ziltoid'}]
result = db.profiles.insert_many(user_profiles)
```

The index prevents us from inserting a document whose user_id is already in the collection:

[27]:
```
new_profile = {'user_id': 213, 'name': 'Drew'}
duplicate_profile = {'user_id': 212, 'name': 'Tommy'}
result = db.profiles.insert_one(new_profile)  # This is fine.
#result = db.profiles.insert_one(duplicate_profile)   ## Try to run this and␣
↪you should get an exception!
```

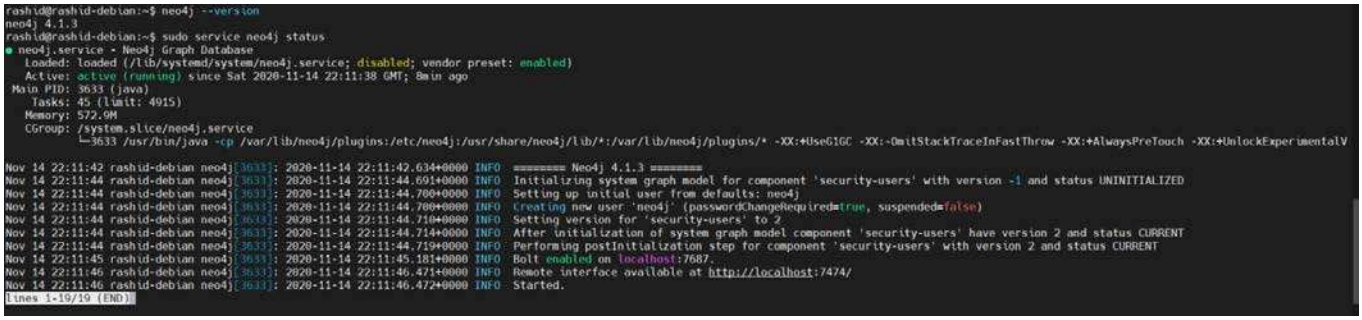
Install Neo4j

```
wget -O - https://debian.neo4j.com/neotechnology.gpg.key | sudo apt-key add -
echo 'deb https://debian.neo4j.com stable latest' | sudo tee -a /etc/apt/sources.list.d/neo4j.list
sudo apt update
sudo apt install -y neo4j
sudo service neo4j start
```

Check Installation:

neo4j --version

sudo service neo4j status



Edit the file */etc/neo4j/neo4j.conf* using a suitable editor such as vi or nano and uncomment the following lines (remove the #). You will need to run this as superuser (sudo).

sudo nano /etc/neo4j/neo4j.conf

dbms.default_listen_address=0.0.0.0
dbms.security.auth_enabled=false

Enable Neo4j, which will cause it to automatically start when the virtual machine is booted.

sudo systemctl enable neo4j

Restart Database

sudo service neo4j restart

Install `neo4j`

The Neo4j Python driver is officially supported by Neo4j and connects to the database using the binary protocol. Use pip to install the driver.

```
pip install neo4j
```

```
(base) C:\Users\rashi>pip install neo4j
Collecting neo4j
  Downloading neo4j-4.1.1.tar.gz (67 kB)
     |████████████████████████████████| 67 kB 1.6 MB/s
Requirement already satisfied: pytz in c:\users\rashi\anaconda3\lib\site-packages (from neo4j) (2020.1)
Building wheels for collected packages: neo4j
  Building wheel for neo4j (setup.py) ... done
  Created wheel for neo4j: filename=neo4j-4.1.1-py3-none-any.whl size=94672 sha256=52c481c0bba1ee3c1b7de2edb04399de0d1eee844f3aca74fd21c6823aea1701
  Stored in directory: c:\users\rashi\appdata\local\pip\cache\wheels\47\91\bf\efc7f154c2f0d12fc45817f35aa503ac35cd547091c26e4ff9
Successfully built neo4j
Installing collected packages: neo4j
Successfully installed neo4j-4.1.1

(base) C:\Users\rashi>
```

The database can also be accessed through a web browser at: http://192.168.56.30:7474/