MongoDB ¶

Introduction

In this lesson, we'll learn about the popular NoSQL database ***MongoDB***, including how to install it on our machine, connect to a mongo database, add how to use it to ***C***reate / ***R***ead / ***U***pdate / ***D***elete (**CRUD**) data!

Objectives

You will be able to:

- Create a MongoDB database
- Insert data into a MongoDB database
- Read data from a MongoDB database
- Update data in a MongoDB database

Installing and Running MongoDB

This part is easy -- in order to install mongoDB, we'll use our favorite package manager, `conda`! This part works the same, regardless of what operating system you're running.

To install MongoDB on your machine, open a terminal or `conda` command prompt and just type:

```
conda install mongodb
```

Next, we have to create a directory to store our Mongo data files:

```
sudo mkdir -p /data/db
```

Give the directory the correct permission:

```
sudo chown -R `id -un` /data/db
```

Now we're ready to run our server! In that same command prompt, just type `mongod`. You'll see the server start up instantly. Note that you must leave this terminal process running in order to make use of the mongoDB instance, so you'll need to leave this one alone, and open a new terminal or command prompt window when you need it.

Examining a Sample MongoDB Record

As we learned in the previous lesson on NoSQL databases, MongoDB is a ***Document Store***. Each record is a JSON document, which can in turn contain other JSON documents in a nested format.

For intance, here's an example record we might see in a MongoDB database:

```
{
   _id: ObjectId(8af37bd7891c)
   title: 'MongoDB Lab',
   description: 'Introductory lab on how to use MongoDB',
   by: 'Flatiron School',
   topics: ['mongodb', 'database', 'NoSQL', 'JSON'],
   sections: [
      {
         section:'Introduction',
         dateCreated: new Date(2019,3,1),
         reading_time_minutes: 1
      },
      {
         user:'Installing and Running MongoDB',
         dateCreated: new Date(2019,3,1),
         reading_time_minutes: 5
      }
      {
         user:'Examining a Sample MongoDB Record',
         dateCreated: new Date(2019,3,1),
         reading_time_minutes: 8
      }
   ]
}
```

This is more common than you might realize -- if you open this file, `index.ipynb`, in a text editor, you'll see that each Jupyter Notebook is structured the exact same way, with each cell and everything it contains being embedded in the larger overall document!

Since we already have experience working with JSON, we'll see that it will actually be quite easy to work with data in mongoDB! Let's explore how to execute *CRUD* operations in MongoDB.

Connecting to the MongoDB Database

To start the MongoDB server and connect to our database, we can just type `mongod` in the terminal or conda command prompt. This will create an instance of a MongoDB server containing our mongo database.

We're now connected to the mongo server. Next, we need to access this server's shell. To do this, open a new terminal window and type `mongo`. Now, you are inside your mongo server and can work with mongo files (databases). To get started, we'll look at what databases this server has access to. Type the command `db` -- you should see a database called `test`.

The mongodb server that's running comes with a host of built-in functions that we can use. To see a full list of the functions for the mongo server itself, we can type `db.help()`. Similarly, the databases themselves have a host of built-ins and functions accessible to us to do things like count the total number of documents in the database and other tasks that might be useful. To see those, we can access the documentation for the `test` database by typing `db.test.help()`

Try doing both of these now, and look around for a minute to familiarize yourself with some of the things that both the server and the database are capable of.

Next, we'll use some of the built-in functions in the `test` database to create and access different records.

Creating, Reading, Updating, and Deleting Information in MongoDB

Typically, the main way you'll be working with MongoDB is through a Python library called `pymongo` that allows us to connect to and manipulate mongo databases in our code, just like `sqlite3` allowed us to connect to and work with SQLite databases in the last section. However, before we do that, it's worth taking a quick look at how we can add data directly to the database through the command line with the running mongo server instance.

Type the following in the terminal running the mongo instance:

```
db.test.save( { a: 1 } )
```

If you have trouble getting this line of code to work, make sure the spacing is exactly as seen in the example above -- each unique character inside the method call is separated by a space, with the exception of the colon, which comes right after the key we want to use.

When you run this line correctly, you'll see the database return a key-value pair showing `_id` as the key and a long, jumbled string of numbers and letter that act as the unique ID for the document we just inserted as the value.

To see a full list of every document stored in the database, we can use `db.test.find()` -- run this now to see the key-value record for the document we created and inserted above. Note that this isn't really practical for anything other than toy databases. For any real work we'll be doing with a mongo database, we'll doing it through Python rather than directly on the mongo server itself. Let's take a look at how we can write queries or do CRUD operations in Python with the `pymongo` library!

Working with Mongodb through Python with `pymongo`

Connecting to mongodb through a python library is going to feel very similar to the boilerplate code we had to use to connect to a SQLite database. To connect to our mongo server through python, we have to:

1. Import the `pymongo` library.
2. Create a client that is connected to our running mongodb server by using the `pymongo` library's `MongoClient` object and passing it the URL for the server (which the mongo server told us as output when we started it up at the very beginning).
3. Get the database that we'll be working with from the `myclient` object -- this can include creating a new database by passing in it's name as a key, just as if we were trying to get it from a Python dictionary.

We'll do this now in the cells below as an example.

In [15]:
```
import pymongo
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_61/2944501792.py in <module>
----> 1 import pymongo

ModuleNotFoundError: No module named 'pymongo'
```

In [ ]:
```
myclient = pymongo.MongoClient("mongodb://127.0.0.1:27017/")
mydb = myclient['example_database']
```

Note that we can get a full list of the names of every database we have by running our clients object's `.list_database_names()` method. However, if we run this method in the cell below, we'll see that the database does not yet exist.

In [ ]:
```
print(myclient.list_database_names())
```

This is because mongoDB doesn't actually create the new database until we have stored some data in it. The act of not doing something until absolutely necessary because another operation needs it is a programming concept called ***lazy execution***. Since our `example_database` database doesn't contain any data yet, mongo hasn't created it yet, so it doesn't show up in the output of our `.list_database_names()` call.

Just as a SQL database has tables, a mongo database has ***Collections*** of documents. We can get a collection or create a new one by passing its name to the database object we created, just like when we passed the database name to the client object.

In the cell below, we create a sample collection.

In [16]:
```
mycollection = mydb['example_collection']
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_61/429258545.py in <module>
----> 1 mycollection = mydb['example_collection']

NameError: name 'mydb' is not defined
```

Now we have a mongo database that contains a collection, but we still haven't added any data to it yet. We can get collection names by using `mydb.list_collection_names()` , but since we haven't added any data to it yet, we'll see that that hasn't been created yet either.

Let's add some data to our database and see what we can do with it.

CRUD Operations with `pymongo`

To insert a document (in SQL, we would call this a *record*) into a mongoDB collection, we make use of the collection's `.insert_one()` method, and pass in the information we want saved as a Python dictionary.

In [13]:
```
example_customer_data = {'name': 'John Doe', 'address': '123 elm street', 'age':

results = mycollection.insert_one(example_customer_data)
results
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_61/438670152.py in <module>
      1 example_customer_data = {'name': 'John Doe', 'address': '123 elm stree
t', 'age': 28}
      2
----> 3 results = mycollection.insert_one(example_customer_data)
      4 results

NameError: name 'mycollection' is not defined
```

When we insert something into mongo, we get back a `results` object. This object contains the unique `_id` of the object we just inserted inside its `.inserted_id` attribute.

In [14]:
```
results.inserted_id
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_61/2640024903.py in <module>
----> 1 results.inserted_id

NameError: name 'results' is not defined
```

If we want to insert 2 or more items at the same time, we can just store the dictionary for each separate record we want to insert in a list, and then every item in that list by passing it to the collection object's `.insert_many()` method.

In [ ]:
```
customer_2 = {'name': 'Jane Doe', 'address': '234 elm street', 'age': 7}
customer_3 = {'name': 'Santa Claus', 'address': 'The North Pole', 'age': 547}
customer_4 = {'name': 'John Doe jr.', 'address': '', 'age': 0.5}

list_of_customers = [customer_2, customer_3, customer_4]

results_2 = mycollection.insert_many(list_of_customers)
```

In [ ]:
```
results_2.inserted_ids
```

Note that we are allowed to assign the unique id for each new document ourselves by just including the key `_id` and the value we want to assign as that document's id. However, in general, it is a best practice to let the database create the unique keys for each document itself,

and to leave that part alone.

Querying data in Python

This is the most important thing we can know how to do in MongoDB -- actually get the data we need!

The quickest and easiest way to get data from a collection is to use the collection object's `.find()` method!

The following cell contains an example of how we can use this to get all the records from our collection.

In [ ]:
```
query_1 = mycollection.find({})
for x in query_1:
    print(x)
```

In the cell above, we grabbed every field from every item in the entire collection. There are times where this is probably too much data for it to be useful for us.

So what if we want to get all the names and addresses for each customer, but not the age? There are two ways we can do this. The first is by passing in a dictionary specifying the fields we want, like so:

In [ ]:
```
query_2 = mycollection.find({}, {'_id': 1, 'name': 1, 'address': 1})
for item in query_2:
    print(item)
```

In this method, we created a dictionary with the key of every item we want, and a `1` as the value to make clear that we want that field returned.

Conversely, if we'd rather specify the key-value pairs we *don't* want returned, we can do that too. All we have to do is create a dictionary containing the keys we don't want returned, and set the value for each to `0`, like so:

In [ ]:
```
query_3 = mycollection.find({}, {'age': 0})
for item in query_3:
    print(item)
```

Note that we can't use both methods at the same time. We have to either specify what we do want, and make sure that every value is a 1, or specify what we don't want, and make sure every corresponding value is a 0. If we pass in a dictionary containing both, we'll get an error in return.

The one exception to this is the `'_id'` key -- we can set that to 0 or 1 to say if we do or don't want it included. This single value does not need to be the same as all the others in the dictionary.

Filtering Query Results

Obviously, we'll rarely want to get all the records at once. There will be many more times where we'll need to get a single record, or to filter records according to their values. Mongo makes these both easy.

For instance, if we know the value for a given key, we can pass that key-value pair (or pairs) into `.find()` as a dictionary, and the results will contain the entire document.

In [ ]:
```
query_4 = mycollection.find({'name': 'Santa Claus'})
for item in query_4:
    print(item)
```

We can also filter queries by using ***Modifiers***. For instance, let's say we wanted to get record for every person in our collection older than 20. We can signify this with the 'greater than' modifier, `"$gt"` and pass in the corresponding value.

In [ ]:
```
query_5 = mycollection.find({"age": {"$gt": 20}})
for item in query_5:
    print(item)
```

We can even pass in ***Regular Expressions*** to filter text data and pattern match! You don't know regular expressions yet, but you will learn them in a few sections. When you do, we encourage you to try using them within a mongodb query!

Updating Documents

Updating a document in Mongo is just as simple as creating or reading data -- there's a method for that! Updating a record works a bit like filtering a query and getting a record with a specific value, although we also pass in an additional dictionary as the second parameter. This second parameter will contain the modifier `'$set'` as the key, and a dictionary containing the key-value pair we want to update. See the example in the cell below:

In [ ]:
```
record_to_update = {'name' : 'John Doe'}
update_1 = {'$set': {'age': 29}}
update_2 = {'$set': {'birthday': '02/20/1986'}}

mycollection.update_one(record_to_update, update_1)
mycollection.update_one(record_to_update, update_2)
query_6 = mycollection.find({'name': 'John Doe'})
for item in query_6:
    print(item)
```

In the cell above, the first update we make updates the value for a key that already exists in the document. The second update adds an entirely new key-value pair to the document. As we can see, the syntax for both is the same (just like when we work with Python dictionaries), and is very easy and intuitive to use.

Deleting Records

We can delete records by using the collection object's `.delete_*()` methods. Just like for inserting records, you'll find that we can use `delete_one()` for a single deletion, or `delete_many()` for multiple deletions.

Let's try deleting the record for `'John Doe'`:

In [ ]:
```
deletion_1 = mycollection.delete_one({'name': 'John Doe'})
print(deletion_1.deleted_count)
```

Note that we can also use modifiers here, too! For instance, in the cell below, we'll delete all records for customers younger than 10.

In [ ]:
```
deletion_2 = mycollection.delete_many({'age': {'$lt': 10}})
print(deletion_2.deleted_count)
```

In [ ]:
```
query_6 = mycollection.find({})
for item in query_6:
    print(item)
```

In [ ]:
```
mycollection.delete_many({})
```

Note: Be *very careful* when using `delete_many()` -- passing in an empty dictionary will delete the entire collection -- it is the mongoDB equivalent of `DROP TABLE`, and can really ruin your day if you aren't careful!

Summary

In this lesson, we learned about how to get mongoDB up and running on our machine, how to connect to it, and how to do some basic CRUD tasks with Python!