Using Python to Interact with MongoDB

This notebook demonstrates basic functioality of MongoDB by way of the **pymongo** library. As it's name implies, pymongo is the MongoDB library for Python, and its **documnentation** can be found here: https://pymongo.readthedocs.io/en/stable/index.html

1.0. Prerequisites

1.1. First, you must install the *pymongo* libary into your *python* environment by executing the following command in a *Terminal window*

- python -m pip install pymongo[srv]

1.2. Next, as with all Jupyter Notebooks, you need to **Import** the libaries that you'll be working with in the notebook,

In [1]:
```
import os
import datetime
import pymongo
import pprint
import pandas as pd
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 3
      1 import os
      2 import datetime
----> 3 import pymongo
      4 import pprint
      5 import pandas as pd

ModuleNotFoundError: No module named 'pymongo'
```

2.0. Connecting to the MongoDB Instance

In [ ]:
```
host_name = "localhost"
port = "27017"

atlas_cluster_name = "smple_mflix"
```

```
atlas_default_dbname = "smple_mflix"
atlas_user_name = "ds2002"
atlas_password = "UVA1819"
```

In [ ]:
```
conn_str = {
    "local" : f"mongodb://{host_name}:{port}/",
    "atlas" : f"mongodb+srv://{atlas_user_name}:{atlas_password}@{atlas_cluster_name}.zibbf.mongodb.net/{atlas_default_
}

client = pymongo.MongoClient(conn_str["atlas"])

print(f"Local Connection String: {conn_str['local']}")
print(f"Atlas Connection String: {conn_str['atlas']}")
```

3.0. Creating Databases, Collections, and Documents

MongoDB creates objects lazily. In other words, databases and collections (somewhat equivalent to a table) are only created on the server when the first document (equivalent to a row or record) is inserted.

In [ ]:
```
db_name = "blog"

db = client[db_name]
client.list_database_names()
```

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

Here we see that even though we're referencing a new database named **blog**, it isn't returned when we query the server for the databases it's serving.

Now let's create a new collection called **posts** by inserting one new **document** using the **insert_one( )** function. Notice that the **document** being inserted is structured similarly to a Python **dictionary**. This is no accident! Both make use of **JavaScript Object Notation (JSON)**. If you pay careful attention, you'll notice that a one-to-many relationship has been modeled by *nesting* related entities within a **List**. Here, the relationship between one **author** and many **tags** has been modeled. We've also inserted a Python-native **datetime** value into the document. This works because MongoDB is actually based on **Binary JavaScript Object Notation (BSON)**, an interchange format created by the developers of MongoDB. Like JSON, BSON supports the embedding of documents and arrays within other documents and arrays; however, BSON also contains extensions that allow representation of data types that are not part of the JSON specification. You can learn more about BSON at: https://bsonspec.org/

In [ ]:
```
#This db is blog, I need post. Don't need to make blogs b/c it makes it easier on the developer, it will make it for yo
post = {"author": "Mike",
        "text": "My first blog post!",
        "tags": ["mongodb", "python", "pymongo"],
        "date": datetime.datetime.utcnow()
       }
# don't care ab how big, schema, etc.
posts = db.posts
post_id = posts.insert_one(post).inserted_id
#Inserted one post in here and then return document ID

print("Document ID: ", post_id)
#returns back document ID, this is unique to each user.
```

Now when we query the client for lists of the databases & collections on the server we see our new database **blog**, and our new collection **posts**.

In [ ]:
```
print("Databases: ", client.list_database_names()) #will add new db 'blog' that we created in the code cell above
print("Collections: ", db.list_collection_names())
```

4.0. Querying MongoDB

Of course the next thing we'll be interested in, is to query the **collection**. Here we retrieve the document we just **inserted**. You may notice that we're not really specifying a query, but because there is only one document in the **collection** it will be returned anyway. If there had been no documents in the **collection** then the result would have been **None**. We're also makine use of the **pprint** (pretty print) library to format our results so they're easily readable.

In [ ]:
```
pprint.pprint(posts.find_one({}))
```

Of course it's possible to **insert** more than one **document** at a time. This is achieved by placing the **documents** into a Python **List**, and then passing them to the **insert_many( )** function. What's more, because MongoDB is designed to support *polyschematism* the new documents we insert aren't required to have matching structures (schemas). Notice that the first document below has no **title** element, and the second document has no **tags** element.

In [ ]:
```
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)
pprint.pprint(result.inserted_ids)
```

Now it's possible to query for specific documents by using JSON **documents** or even simple **key : value** pair notations (which are actually simple JSON documents). First, using the **find_one( )** method the first occurance that matches the specified criterea will be returned. To ensure you get exactly the **document** you want, you can use its ObjectID.

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

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

It's also possible to iterate over multiple **documents** by way of the **find( )** method, which returns a cursor containing references to multiple documents. The MongoDB equivalent of the SQL query **SELECT * FROM posts** is achieved by calling the **find( )** function with no argument at all, and he MongoDB equivalent of **SELECT * FROM posts WHERE author = 'Mike'** is achieved by passing the simple JSON document **{"author" : "Mike"}**.

In [ ]:
```
for post in posts.find():
    pprint.pprint(post)
#iterates through the list
```

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

The number of documents in a collection, or the number of documents that match a set of criterea, can be retrieved using the **count_documents( )** function.

In [ ]:
```
print("All Docs: ", posts.count_documents( {} ))
print("Matching Docs: ", posts.count_documents( {"author" : "Eliot"} )) #how many docs match this
```

Many advanced querying techniques can be achieved using MongoDB. For example, the following **range query** retrieves all documents older than *November 12, 2009*, sorted by *author*. The equivalent SQL query would be **SELECT * FROM posts WHERE date**

< '2009-11-12:12.0.0:00' ORDER BY author.

Notice here that in order to specify the range **less than**, the special operator **$lt** was used, and that the comparison was nested within curly braces.

In [ ]:
```
d = datetime.datetime(2009, 11, 12, 12)

for post in posts.find({"date": {"$gt": d}}).sort("author"):
    pprint.pprint(post)
```

5.0. Indexes, Unique Constraints and Primary Keys

Also equivalent to relational database management systems are the use of **indexes** to expedite data retrieval, and to enforce **uniqueness** where desired. When designing RDBMS tables, it is customary to create a **Primary Key** that uniquely identifies each observation (row). By default, MongoDB creates an index on the **_id** field, but it may be desireable to enforce uniqueness on user-defined values such as we have seen with **customer_id, employee_id, product_id,** and **shipper_id**. To that affect, the following code creates an *unique* index on the *user_id* element that is sorted in *ascending* order.

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

Now, we can insert some new documents that leverage the new **user_id** unique key index...

In [ ]:
```
jedi_profiles = [
    {'jedi_id': 211, 'name': 'Luke'},
    {'jedi_id': 212, 'name': 'Yoda'}]

result = db.profiles.insert_many(jedi_profiles)
print(result)
```

... but if we attempt to insert a record having a preexisting *user_id* then a **Duplicate Key error** will be thrown.

In [ ]:
```
sith_profile = {'jedi_id': 212, 'name': 'Anakin'}
result = db.profiles.insert_one(sith_profile)
```

In [ ]:
```
for profile in db.profiles.find():
    pprint.pprint(profile)
```

6.0. Dropping Databases and Collections

Of course what can be created can also be destroyed. Here are the **pymongo** methods for dropping **collections** and **databases**.

First, if you drop the last, or only, collection in a database then the entire database will be dropped as well... so first we'll create a second collection named **users** so we can demonstrate the methods for dropping collections and databases.

In [ ]:
```
user = {"first_name" : "John",
        "last_name" : "Doe",
        "role" : "administrator"
       }

users = db.users
user_id = users.insert_one(user).inserted_id

print("User ID: ", user_id)
print("Databases: ", client.list_database_names())
print("Collections: ", db.list_collection_names())
```

Here we see that we've created a new **user** and subsequently a new collection **users**. Now let's go ahead and drop the **posts** collection.

In [ ]:
```
for c in db.list_collection_names():
    db.drop_collection(c)

print("Databases: ", client.list_database_names())
print("Collections: ", db.list_collection_names())
```

So now we just have the **users** collection in the **blog** database. Now let's go ahead and drop the **blog** database.

In [ ]:
```
client.list_database_names()
```

In [ ]:
```
result = client.drop_database(db_name)

print("Return Value: ", result)
print("Databases: ", client.list_database_names())
print("Collections: ", db.list_collection_names())
```

In [ ]: