Exercise-2

In this exercise, we will be learning how to access MongoDB database from its drivers using a programming language. We will be using python and its MongoDB driver - **PyMongo**.

If PyMongo is not installed already,

`$python -m pip install pymongo` would do it.

PyMongo provides API **`MongoClient()`** to connect to a running 'mongod' instance.

One can use it as follows:

```
from pymongo import MongoClient

client=MongoClient('connectionURL')
```

`ConnectionURL` here can be gotten from atlas site just like the way we did for compass and mongo shell. You just have to choose the option 'connect your application'. Example URL looks like this:

```
mongodb+srv://2017*****:<password>@mycluster-ydqrn
.mongodb.net/test?retryWrites=true&w=majority
```

Now we just make a **`db`** variable referencing the database for ease:

```
db=client['201701***']
```

Now the rest of the operations are almost identical to the shell commands learnt in exercise 1. Some changes are mentioned below:

insertMany() ← insert_many()

updateMany() ← update_many()

And so on…

Documents are conveniently represented by dictionaries in python.

So naturally, these functions take in dictionaries as arguments.

As we have learnt in exercise-1 that the find functions always return a cursor and this needs to be iterated to display the result. You can use a for loop to move the cursor in python.

```
for docs in db.posts.find():

    print(doc)
```

This can also be used for making a list of documents for multiple inserts.

Write answers to questions 1 and 2 in **`/exercise2/soln/id_ex2.txt`**

Schema Validation:

You might have noticed the flexibility MongoDB provides to the user by allowing for documents with any fields in a collection. However, it does also have the option of putting constraints on the structure of documents in a collection. This is done by specifying a **`validator`** field in **`createCollection().`** Of course, this is to be done at the time of the creation of the collection. The value of a validator field can be any query operator except **`$geoNear`**, **`$near`**, **`$nearSphere`**, **`$text`**, and **`$where`**. **`$jsonSchema`** is one such common validator and covers almost all constraints one can put on a JSON document. **`$jsonSchema`** provides several options of which only a few are discussed.

$jsonSchema:

bsonType: Used to specify the type of a given field. Can take values like "object", "array", "int", "long", "double", "boolean", "string", and "null".

required: An array of unique strings which enforces that document contains all the fields in the array.

properties: A valid **`$jsonSchema`** where each value is also a valid **`$jsonSchema`** object. (It is just a recursive definition to allow for the definition of subfields). An example might clarify:

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: [ "name", "year", "major", "address" ],
         properties: {
            name: {
               bsonType: "string",
               description: "must be a string and is required"
            },
            year: {
               bsonType: "int",
               minimum: 2017,
               maximum: 3017,
               description: "must be an integer in [ 2017, 3017 ] and is required"
            },
            major: {
               enum: [ "Math", "English", "Computer Science", "History", null ],
               description: "can only be one of the enum values and is required"
            },
            gpa: {
               bsonType: [ "double" ],
               description: "must be a double if the field exists"
            },
            address: {
               bsonType: "object",
               required: [ "city" ],
               properties: {
                  street: {
                     bsonType: "string",
                     description: "must be a string if the field exists"
                  },
                  city: {
                     bsonType: "string",
                     "description": "must be a string and is required"
                  }
               }
```

Notice that the ‘properties’ field must have all the names in the ‘required’ array field as its subfields but can have additional fields too. The bsonType for a document is ‘object’. It is therefore used at the top-level bsonType field always.

Question 1: Write a createCollection command to create ‘sales’ collection, according to the description given below.

Each document describes a sales transaction at a company. ‘salesDate’ (timestamp), ‘purchaseMethod’, ‘storeLocation’, information of whether a coupon is used or not through ‘couponUsed’(true/false) is recorded for each transaction. Each transaction has a list of items with their ‘price’(decimal), ‘quantity’(integer) and of course ‘name’ recorded. The company has stores only at the following locations at the moment:

- Seattle
- New York
- Austin
- London
- Denver
- San Diego

and supports purchases through ‘Online’, ‘In store’, ‘Phone’ methods only.

The items available for purchase are scattered across the following tags:

- kids
- school
- stationary
- writing
- organization
- travel

- electronics
- office
- general

Each item has a list of tags also recorded within a transaction. The details of the customer performing the transaction like email, ‘age’(integer) are recorded. However, the customer may also provide additional, non-mandatory details like ‘gender’(M/F/other) and ‘satisfaction’(integer from 1 to 5 both inclusive).

Import and Export Data:

MongoDB provides ways to import/export data from external resources (from JSON and CSV files). **`mongoimport`** can be used from your system shell or command prompt to import data.

```
mongoimport --host <host> --port <port> --db
<databasename> --collection <collectionname>
--authenticationDatabase <authdbname> --username
<user> --password <password> --drop --file <path
of file relative to current dir>
```

In our case:

```
mongoimport --uri <connection string>
can be used.
where <connection string> is
mongodb+srv://201701***:<password>@mycluster-ydqrn
.mongodb.net
```

notice that this is a part of the connection URL used earlier.

- **`<authdb>`** represents the database on which the user has read, write rights.
- **`<databasename>`** represents the database in which the collection is to be created.

- **`--drop`** drops the collection if it already exists
- **`--jsonArray`** if the document has an array of JSON documents
- The rest is self-explanatory.

Similarly, **`mongoexport`** is available.

```
mongoexport --collection=events --db=reporting
--out=events.json
```

Question 2: In order to set up for this exercise, import data from `/data/sales/sales.json` to your database in the collection named sales. (Check if all the documents pass the schema validation)

Questions 3-X: Complete the following questions on the collection 'sales' created in the previous exercise:

Answers are to be written in the form of programs in .py files provided in **`/exercise2/soln/`**

Boilerplate is provided to you in all the files.

1. Get the total sales transactions made at each store.
2. Get the total sales amount at the Denver store.
3. Make a new collection called stock_replenish in which each document is about an item('name') followed by an array of objects('sales_history') each representing storeLocation(representing where the item was purchased) and quantity(representing the total amount purchased). This would help the company replenish the sold items.