CSE 4410

DATABASE MANAGEMENT SYSTEMS II LAB

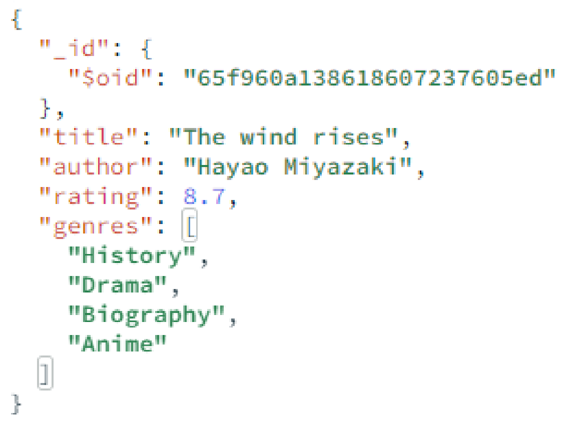
Notes On: MongoDB

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ISLAMIC UNIVERSITY OF TECHNOLOGY

MARCH 27, 2024

1 Getting Started with MongoDB

MongoDB is a document database which is a type of non-relational database. It stores data in a type of JSON format called BSON. A record in MongoDB is a document, which is a data structure composed of key value pairs similar to the structure of JSON objects and and the field values may include numbers, strings, booleans, arrays, or even nested documents. Similar types of documents together create a collection (similar concept of table in relational database).



```
{
  "_id": {
    "$oid": "65f960a138618607237605ed"
  },
  "title": "The wind rises",
  "author": "Hayao Miyazaki",
  "rating": 8.7,
  "genres": [
    "History",
    "Drama",
    "Biography",
    "Anime"
  ]
}
```

Figure 1: A Sample document

Unlike SQL tables, there are no structural limits on what you can store in MongoDB. Data schemas are not enforced: You can store whatever you like wherever you like. This makes MongoDB ideal for more unstructured or messy data structures.

2 Environment Setup

MongoDB can be installed locally or hosted in the cloud. Locally installed MongoDB will allow you to host your own MongoDB server on your hardware. This requires you to manage your server, upgrades, and any other maintenance. However, you can also use MongoDB Atlas, a cloud database platform.

2.1 Local Database

To start working on local MongoDB one needs to install the following files.

1. Install MongoDB Community Server using `mongodb-windows-x86_64-7.0.7-signed.msi`. (While installing choose setup type `complete`. Make sure to check `Install MongoDB as a Service` and also check `Install MongoDB Compass` which is a GUI for interacting with the database).

2. Install MongoDB Shell using `mongosh-2.2.0-x64.msi` which allows to interact with the database from a terminal.

3. In case you forget to check MongoDB Compass, install using `mongodb-compass-1.42.2-win32-x64.exe`.

All the files mentioned above have been provided in Google Classroom.

2.2 Cloud Database

To get started with cloud-based MongoDB, sign up for a free `MongoDB Atlas` account.

1. After creating an account, set up a free "Shared Cluster" and choose the preferred cloud provider and region.

2. Set up a user and add an IP address to the list of allowed IP addresses.
3. Under "Database Access", create a new user and keep track of the username and password.
4. Next, under "Network Access", add your current IP address to allow access from your computer.
5. To use cloud-based database, you need to install either MongoDB Compass or MongoDB Shell.

2.3 Connection Set Up

You can access your MongoDB database by using the following:

1. The MongoDB port (27017 by default in case of local).
2. Use connection string, and user password (in case of cloud).

2.4 Basics

- To open mongoDB shell in command prompt

  ```
  > mongosh
  ```

- To see available operation

  ```
  > help
  ```

- To see available database

  ```
  > show dbs
  ```

- To switch into a a database for further operation

  ```
  > use <database_name>
  ```

- To see details of a database

  ```
  > db.stats()
  ```

- To see the collections of the database

  ```
  > show collections
  ```

- To insert a single data

  ```
  > db.<collection_name>.insertOne({<attribute_name>:<value>,......,
  <attribute_name>:<value>})
  ```

  we don't necessarily to have an existing collection to insert data. If there is no collection with the collection name it will create a new collection and insert the data there. So if we want create a new collection we can simply do this by inserting a new entry.

- To insert an entry with an array

```
> db.<collection_name>.insertOne({<attribute_name>:<value>,......,
<attribute_name>:[<value>,......,<value>]})
```

- To insert an entry with an array of object/ nested entry

```
> db.<collection_name>.insertOne({<attribute_name>:<value>,......,
<attribute_name>:[{<attribute_name>:<value>,......,<attribute_name>:
<value>},{<attribute_name>:<value>,......,<attribute_name>:<value>}]})
```

- To insert many data at a time

```
> db.<collection_name>.insertMany([{<attribute_name>:<value>,......,
<attribute_name>:<value>},.....,{<attribute_name>:<value>,......,
<attribute_name>:<value>}])
```

- To fetch the entries of a collection

```
> db.<collection_name>.find()
```

  This command will show first 20 entries of the collection. To see 20 more entries type -> it.

- To fetch one entry from a collection

```
> db.<collection_name>.findOne()
```

- To fetch entries from a collection with condition

```
> db.<collection_name>.findOne()
```

- To fetch entries with certain attributes

```
> db.<collection_name>.find({<attribute_name>:<value>,......,
<attribute_name>:<value>}{<attribute_name>:1,...,<attribute_name>:1})
```

- To fetch entries with a certain value in the array attribute

```
> db.<collection_name>.find({<array_attribute_name>: <value>})
```

- To fetch entries with multiple certain values in the array attribute

```
> db.<collection_name>.find({<array_attribute_name>: {$all:[<value>,
...,<values>]}})
```

- To fetch entries with that contains only certain values in the array attribute

```
> db.<collection_name>.find({<array_attribute_name>: [<value>]})
```

- To fetch entries with a certain value in the nested attribute

```
> db.<collection_name>.find({"<outer_attribute_name>.<inner_attribute
_name>":<value>})
```

- To delete a document

```
> db.<collection_name>.deleteOne({<attribute_name>:<value>})
```

- To delete many document

```
> db.<collection_name>.deleteMany({<attribute_name>:<value>})
```

- To update a document

```
> db.<collection_name>.updateOne({<attribute_name>:<value>},{$set: {
<attribute_name>:<value>,..., <attribute_name>:<value>}})
```

- To update many document

```
> db.<collection_name>.updateMany({<attribute_name>:<value>},{$set: {
<attribute_name>:<value>,..., <attribute_name>:<value>}})
```

2.5 Method Chaining:

- To count how many entries we have got

```
> db.<collection_name>.find({<attribute_name>:<value>,......,
<attribute_name>:<value>}).count()
```

- To limit the entries we want to get

```
> db.<collection_name>.find({<attribute_name>:<value>,......,
<attribute_name>:<value>}).limit(<value>)
```

- To limit the entries we want to get

```
> db.<collection_name>.find().sort({<attribute_name><1/-1>,...,
<attribute_name>:<1/-1>})
```

to sort in ascending order use 1 and for descending order use -1

2.6 Special Query Operator: (begins with $)

- greater than operator, ($gt)

```
> db.<collection_name>.find({<attribute_name>: {$gt: <value>}})
```

- less than operator, ($lt)
- greater than or equal, ($gte)

- less than or equal, ($lte)
- or operator, ($or)

```
> db.<collection_name>.find({$or: [{<attribute_name>: <value>},
{<attribute_name>: <value>}]})
```

- and operator, ($and)
- in operator, ($in)

```
> db.<collection_name>.find({<attribute_name>:{$in: [<value>,...,
<value>]}})
```

- not in operator, ($nin)
- in operator, ($inc)

```
> db.<collection_name>.updateOne({<attribute_name>:<value>},{$inc: {
<attribute_name>:<value>}})
```

  to increment assign a positive value and for decrement assign a negative value

- pull operator, ($pull)

```
> db.<collection_name>.updateOne({<attribute_name>:<value>},{$pull: {
<array_attribute_name>:<value>}})
```

- push operator, ($push)
- each operator, ($each)

```
> db.<collection_name>.updateOne({<attribute_name>:<value>},{$push: {
<array_attribute_name>:{$each: [<value>,...,<value>]}}})
```

- Counting item of an array, ($size)

```
> db.<collection_name>.find({},{alias: {$size: "$attribute_name"}})
```

- To match an element from an array of object($elemMatch)

```
> db.<collection_name>.find({<attribute_name>:{$elemMatch:
{<nested_attribute_name>:<condition>}}})
```

- To chech whether an element from an array of object exit or not($exists)

```
> db.<collection_name>.find({<attribute_name>:{$elemMatch:
{<nested_attribute_name>:{$exists: <true/false>}}}})
```

2.7 Joining two collections

- To join two collections

```
>db.<collection_name>.aggregate([
    {$lookup:{
        from:"other_table_name",
        localField:"attribute_of_the_current_table",
        foreignField:"attribute_of_the_other_table",
        as:"alias_for_the_other_table"}
    },
    {$unwind: "alias_for_the_other_table"}
])
```

  unwind is used for showing the details of the nested object that appears as an alias.
- If you want to add any condition while joining two collections

```
>db.<collection_name>.aggregate([
    {$match:{<atrribute_name>:condition}},
    {$lookup:{
        from:"other_table_name",
        localField:"attribute_of_the_current_table",
        foreignField:"attribute_of_the_other_table",
        as:"alias_for_the_other_table"}
    }
])
```

- If you want to project some specific attributes while joining two collections

```
>db.<collection_name>.aggregate([
    {$match:{<atrribute_name>:condition}},
    {$lookup:{
        from:"other_table_name",
        localField:"attribute_of_the_current_table",
        foreignField:"attribute_of_the_other_table",
        as:"alias_for_the_other_table"}
    },
    {$project:{<attribute_name>:1,<attribute_name>:1}}
])
```

- If you want to join more than two collections

```
>db.<collection_name>.aggregate([
    {$lookup:{
        from:"other_table_name",
        localField:"attribute_of_the_current_table",
        foreignField:"attribute_of_the_other_table",
        as:"alias_for_the_other_table"}
    },
    {$unwind: "alias_for_the_other_table"},
    {$lookup:{
```

```
                from:"2nd_other_table_name",
                localField:"attribute_of_the_current_table",
                foreignField:"attribute_of_the_2nd_other_table",
                as:"alias_for_the_other_table"}
            },
            {$unwind: "alias_for_the_2nd_other_table"},
            {$project:{<attribute_name>:1,<attribute_name>:1}}
        ])
```

2.8 Using MongoDB Data Validation Schemas

Unlike SQL, data definition schemas are not necessary in MongoDB. You can post any data to any document in any collection at any time.
This provides considerable freedom. However, there may be times when you want to insist that rules are followed. Validation rules can be specified using a $jsonSchema object which defines an array of required items and the properties of each validated field.

- Let's say a collection has already been created, but you can still define a schema:

```
>db.runCommand({
    collMod: "<collection_name>",
    validator: {
        $jsonSchema: {
            required: ["atrribute_name"],
            properties: {
            attribute_name: {
                bsonType: "data_type",
                description: "any description"
                }
            }
        }
    }
});
```

An example is given here,

```
db.runCommand({
    collMod: "person",
    validator: {
        $jsonSchema: {
            required: ["name"],
            properties: {
                name: {
                bsonType:"string",
                description:"name␣string␣required"
                }
            }
        }
    }
});
```

- Schemas can also be defined before creating a collection. The following command implements the same rules as above:

```
>db.createCollection("collection_name", {
    validator: {
        $jsonSchema: {
            required: ["atrribute_name"],
            properties: {
            attribute_name: {
                bsonType: "data_type",
                description: "any description"
                }
            }
        }
    }
});
```

Example,

```
db.createCollection("person", {
    validator: {
        $jsonSchema: {
            required: ["name"],
            properties: {
                name: {
                bsonType:"string",
                description:"name␣string␣required"
                }
            }
        }
    }
});
```

- Example of a more complex validator:

```
db.createCollection("person", {
    validator: {
        $jsonSchema: {
            required: [ "name", "email", "telephone" ],
            properties: {
                name: {
                    bsonType: "string",
                    description: "name␣string␣required"
                },
                email: {
                    bsonType: "string",
                    pattern: "^.+\@.+$",
                    description: "valid␣email␣required"
                },
                telephone: {
                    bsonType: "array",
                    minItems: 1,
                    description: "at␣least␣one␣telephone␣
                       number␣required"
                }
            }
        }
    }
});
```