**Week 10**

**Create and Drop database**

MongoDB use DATABASE\_NAME is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax: use DATABASE\_NAME

>use mydb

switched to db mydb

If you want to check your databases list, use the command show dbs.

>show dbs

local 0.78125GB

test 0.23012GB

MongoDB db.dropDatabase() command is used to drop a existing database.

### Syntax db.dropDatabase()

>use mydb

switched to db mydb

>db.dropDatabase()

>{ "dropped" : "mydb", "ok" : 1 }

**Create and Drop Collections**

 db.createCollection(name, options) is used to create collection.

### Syntax: db.createCollection(name, options)

>use test

switched to db test

>db.createCollection("mycollection")

{ "ok" : 1 }

>show collections

mycollection

system.indexes

db.collection.drop() is used to drop a collection from the database.

Syntax: db.COLLECTION\_NAME.drop()

>db.mycollection.drop()

true

**CRUD Operation on Mongo db**

**Creating a Table.**

db.createCollection("student")

{ ok: 1 }

show tables

student

## insert() Method

To insert data into MongoDB collection, you need to use MongoDB's insert() or save() method.

### Syntax: db.COLLECTION\_NAME.insert(document)

### db.student.insert({"id":1,"name":"chandru","mark":300})

db.student.insertMany([{"id":1,"name":"chandru","mark":300},

{"id":2,"name":"suman","mark":290}])

**View data from Table.**

db.student.find({})

**Update.**

db.student.update({"name":"chandru"},{$set:{"name":"sekar",id:5}})

**Delete only one data.**

db.student.deleteOne({"name":"sekar"})

**Indexing**

db.indexing.createIndex({"name":1})

'name\_1'

**equal.**

db.student.find({name:{$eq:"chandru"}})

**GreaterThan.**

db.student.find({mark:{$gt:300}})

**LessThen.**

db.student.find({mark:{$lt:400}})

**GreaterThen and Lessthen in Between**

db.student.find({mark:{$gt:290,$lt:400}})

**Sort**

db.student.find({},{id:0}).sort({"name":1})

**Count.**

db.student.countDocuments()

db.student.countDocuments({"name":"chandru"})

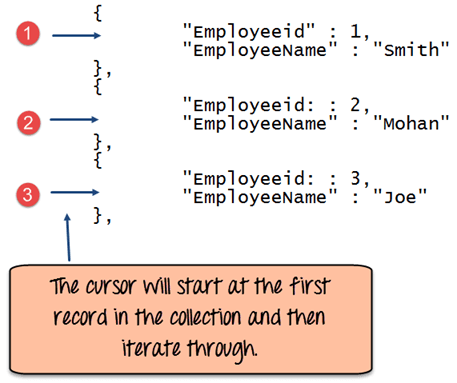
**Limit:**  **limit()** method limits the number of records or documents that you want.

db.student.find().limit(2)

**Cursor**

When the **db.collection.find ()** function is used to search for documents in the collection, the result returns a pointer to the collection of documents returned which is called a cursor.

By default, the cursor will be iterated automatically when the result of the query is returned.



**Indexing**

* The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

## The createIndex() Method

### Syntax: createIndex()

>db.COLLECTION\_NAME.createIndex({KEY:1})

* Here key is the name of the field on which you want to create index and 1 is for ascending order.
* To create index in descending order you need to use -1.

>db.mycol.createIndex({"title":1})

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"ok" : 1

}

In **createIndex()** method you can pass multiple fields, to create index on multiple fields.

>db.mycol.createIndex({"title":1,"description":-1})

## The dropIndex() method

### Syntax: DropIndex()

>db.COLLECTION\_NAME.dropIndex({KEY:1})

Here, "key" is the name of the file on which you want to remove an existing index.

You can also specify the name of the index directly as:

dropIndex("name\_of\_the\_index")

> db.mycol.dropIndex({"title":1})

{

"ok" : 0,

"errmsg" : "can't find index with key: { title: 1.0 }",

"code" : 27,

"codeName" : "IndexNotFound"

}

## The dropIndexes() method

This method deletes multiple (specified) indexes on a collection.

### Syntax: db.COLLECTION\_NAME.dropIndexes()

> db.mycol.createIndex({"title":1,"description":-1})

>db.mycol.dropIndexes({"title":1,"description":-1})

{ "nIndexesWas" : 2, "ok" : 1 }

## The getIndexes() method

This method returns the description of all the indexes int the collection.

### Syntax: db.COLLECTION\_NAME.getIndexes()

> db.mycol.getIndexes()

[{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_",

"ns" : "test.mycol"

},

{

"v" : 2,

"key" : {

"title" : 1,

"description" : -1

},

"name" : "title\_1\_description\_-1",

"ns" : "test.mycol"

}]

**Aggregation**

* Aggregations operations process data records and return computed results.
* Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.
* In SQL count(\*) and with group by is an equivalent of MongoDB aggregation.

## The aggregate() Method

### Syntax: aggregate()

>db.COLLECTION\_NAME.aggregate(AGGREGATE\_OPERATION)

{

\_id: ObjectId(7df78ad8902c)

by\_user: 'tutorials point',

url: 'http://www.tutorialspoint.com',

likes: 100

},

{

\_id: ObjectId(7df78ad8902d)

by\_user: 'tutorials point',

url: 'http://www.tutorialspoint.com',

likes: 10

},

|  |  |  |
| --- | --- | --- |
| **Expression** | **Description** | **Example** |
| $sum | Sums up the defined value from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$sum : "$likes"}}}]) |
| $avg | Calculates the average of all given values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$avg : "$likes"}}}]) |
| $min | Gets the minimum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$min : "$likes"}}}]) |
| $max | Gets the maximum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$max : "$likes"}}}]) |
| $push | Inserts the value to an array in the resulting document. | db.mycol.aggregate([{$group : {\_id : "$by\_user", url : {$push: "$url"}}}]) |
| $addToSet | Inserts the value to an array in the resulting document but does not create duplicates. | db.mycol.aggregate([{$group : {\_id : "$by\_user", url : {$addToSet : "$url"}}}]) |
| $first | Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage. | db.mycol.aggregate([{$group : {\_id : "$by\_user", first\_url : {$first : "$url"}}}]) |
| $last | Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage. | db.mycol.aggregate([{$group : {\_id : "$by\_user", last\_url : {$last : "$url"}}}]) |

**Perform CRUD Operations on MongoDB through REST API using Spring Boot Starter Data MongoDB**

**Step 1:** Create a Spring Boot project.

**Step 2:** Add the following dependency

* Spring Web
* MongoDB
* Lombok
* DevTools

**Step 3:** Create 3 packages and create some classes and interfaces inside these packages

* entity
* repository
* controller

**Step 4:** Inside the entity package create a Book.java file.

package com.example.security.entity;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

@Data

@NoArgsConstructor

@AllArgsConstructor

@Document(collection = "Book")

public class Book {

@Id

private int id;

private String bookName;

private String authorName;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getBookName() {

return bookName;

}

public void setBookName(String bookName) {

this.bookName = bookName;

}

public String getAuthorName() {

return authorName;

}

public void setAuthorName(String authorName) {

this.authorName = authorName;

}

}

**Step 5:**Inside the repository package

Create a simple interface and name the interface as **BookRepo**. This interface is going to extend the **MongoRepository**

package com.example.security.repository;

import org.springframework.data.mongodb.repository.MongoRepository;

import com.example.security.entity.Book;

public interface BookRepo

extends MongoRepository<Book, Integer> {

}

**Step 6:**Inside the controller package. Inside the package create one class named as **BookController**

package com.example.security.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import com.example.security.entity.Book;

import com.example.security.repository.BookRepo;

import java.util.List;

@RestController

public class BookController {

@Autowired

private BookRepo repo;

@PostMapping("/addBook")

public String saveBook(@RequestBody Book book){

repo.save(book);

return "Added Successfully";

}

@GetMapping("/findAllBooks")

public List<Book> getBooks() {

return repo.findAll();

}

@DeleteMapping("/delete/{id}")

public String deleteBook(@PathVariable int id){

repo.deleteById(id);

return "Deleted Successfully";

}

}

**Step 7:**Below is the code for the application.properties file

server.port:8989

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017

spring.data.mongodb.database=jss

**Step 8:** Inside the MongoDB Compass

Go to your MongoDB Compass and create a Database named **BookStore**and inside the database create a collection named **Book**

**Testing the Endpoint in Postman**

POST – <http://localhost:8989/addBook>

GET – <http://localhost:8989/findAllBooks>

DELETE – http://localhost:8989/delete/1