**MongoDB**

MongoDB is a popular open-source NoSQL database management system that provides a flexible and scalable approach to handling and organizing data. It falls under the category of document-oriented databases, and its design philosophy emphasizes scalability, flexibility, and ease of development.

**Install MongoDB:**

* Run the installer.
* Follow the installation wizard's instructions.
* Choose the "Complete" setup type for a full installation.
* MongoDB will be installed in the C:\Program Files\MongoDB\Server\ directory by default.
* Navigate to Edit Environment variables.
* Go to Path.
* Add this Path C:\Program Files\MongoDB\Server\7.0.
* Open another terminal or command prompt.
* Connect to the MongoDB server using the MongoDB shell: mongo.
* You are now in the MongoDB shell, and you can start interacting with the MongoDB server.

**Key Concepts:**

* Document-Oriented:

MongoDB stores data in flexible, JSON-like documents called BSON (Binary JSON). Documents can have nested structures and support arrays, making it easy to represent complex data.

* Collections:

Documents are organized into collections, which are analogous to tables in relational databases. Collections group related documents and allow for efficient querying.

* Schema-less:

MongoDB is schema-less, meaning that documents in the same collection can have different fields. This flexibility enables developers to evolve data models without requiring a predefined schema.

* Indexes:

MongoDB supports the creation of indexes on fields, improving query performance. Indexes can be created on single fields, compound fields, and arrays.

* Query Language:

MongoDB uses a powerful and expressive query language that allows developers to perform a wide range of queries, including filtering, sorting, and aggregation.

* Scalability:

MongoDB is designed to scale horizontally, allowing for the distribution of data across multiple servers or clusters. This ensures high availability and performance as data grows.

* Replication:

MongoDB supports replica sets, providing data redundancy and fault tolerance. In a replica set, one primary node handles writes, while secondary nodes replicate data for read scalability and failover.

* Sharding:

Sharding is a method used to distribute data across multiple servers or clusters to handle large amounts of data and traffic. MongoDB's sharding capability supports horizontal scaling.

* Aggregation Framework:

MongoDB includes a powerful aggregation framework for performing data transformations and analysis within the database.

* Transactions:

MongoDB supports multi-document transactions, allowing for atomic operations on multiple documents. This ensures data consistency in complex operations.

**Use Cases:**

* Web Applications:

MongoDB is well-suited for web applications with dynamic and evolving schemas.

* Real-time Big Data Analytics:

Its ability to handle large volumes of data and support for real-time analytics makes MongoDB suitable for big data applications.

* Content Management Systems:

MongoDB's flexible schema and document-oriented nature make it suitable for content management systems.

* Internet of Things (IoT):

MongoDB's scalability and ability to handle diverse data types make it a good fit for IoT applications.

**MongoDB Basic Commands**

* Connect to MongoDB

mongo

* Show all databases:

show dbs

* Switch to a specific database:

Show dbs

* Show collections in the current database:

show collections

* Create the collection

db.createCollection(“ALT”)

* Insert the document into collection

db.<collection\_name>. insert({ key: value, key2: value2, ... })

db.ALT.insert({“name”:”seamount”})

* Query with criteria:

db.<collection\_name>. find({ key: value })

If there are no criteria, then db.ALT.find()

* Drop all the tables.

db.ALT.drop()

* Pretty print the results:

db.ALT.find(). pretty

* Update the document.

db.ALT.update({“course”:”java”}, {$set: {“course”:”python”}})

* Delete documents:

db.<collection\_name>. remove({ key: value })

* Remove all the documents

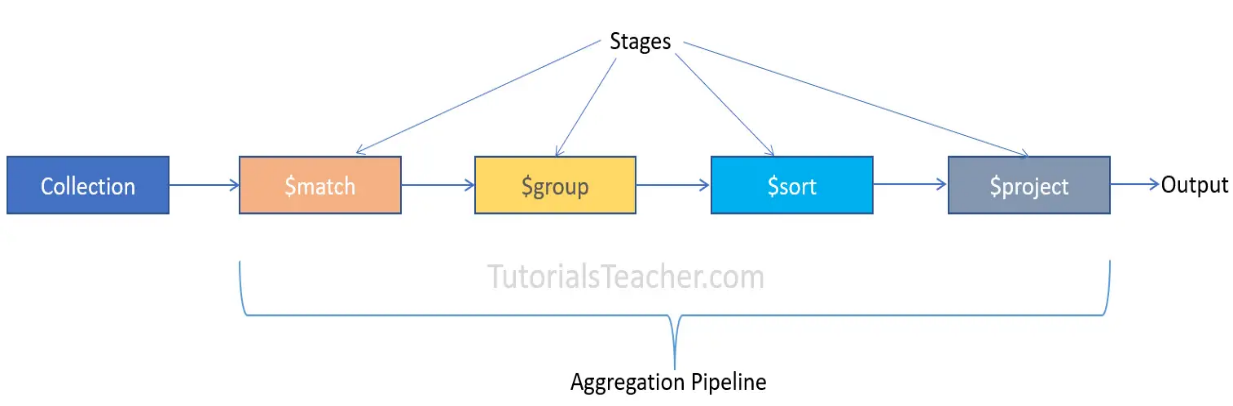
db.ALT.remove({})

* Create an index:

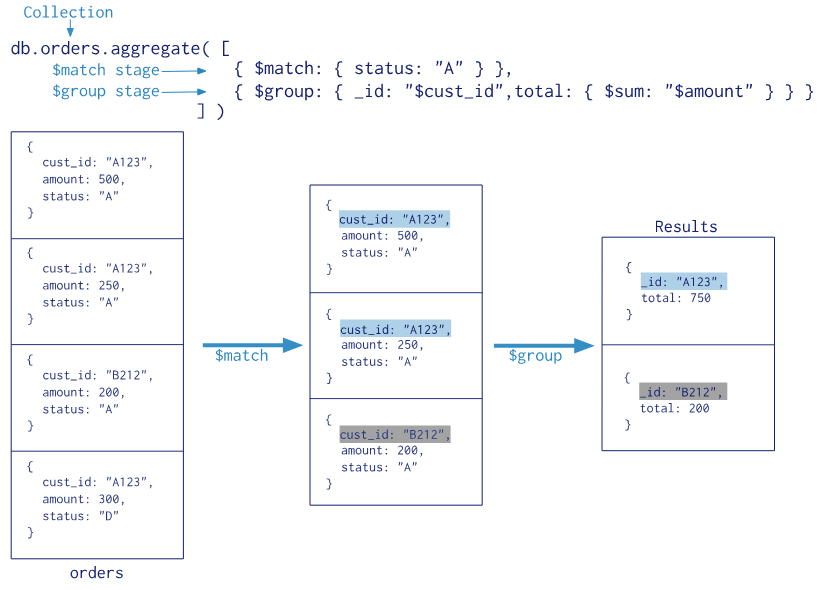
db.<collection\_name>. createIndex({ key: 1 })

**MongoDB Aggregation**

Aggregation is the process of selecting data from a collection to process multiple documents and returns computed results



db. orders. aggregate({$match:{status:”A”}},{$group:{id:”$cust\_id”},total:{$sum:”$amount”}}



There are some aggregation operations similar to sum.

$AVG, $MIN,$MAX,

$PUSH -> Adds the values to an array in the associated document.

**Aggregation Stages in the code**

GroupOperation(GroupBy)

MatchOpearation(Condition)

Operation (Sort)

Aggregation operation=Aggregation.new Aggregation(GroupOperation,MatchOperation,Operation)

Example

//MatchOperation

MatchOperation matchOperation =Aggregation.match(new Criteria(“age”).is(Age));

//SortOperation

SortOperation sortOperation=Aggregation.sort(Sort.by(Sort.Direction.DESC,”age”));

Aggregation aggregation=Aggregation.newAggregation(matchOperation,sortOperation);

AggregationResults output =mongoTemplte.aggregate(aggregate,”employee”,Employee)

**Comparison Operators**

$eq-> equals

$ne-> notequals

$gt->greaterthan

$gte->greaterthan or equalto

$lt->lowerthan

$lte->lowerthan

$neq-> notequalto

Examples

Db.orders.find({“paymentMode”:{$eq:”Card”}})

Db.orders.find({“total”:{$nq:600}})

Db.orders.find({“total”:{$gt:600}})

**Logical Operators**

$and->And Operator

$or-> Or Operator

$nor-> Nor Operator

$not->Not Operator

Db. orders.find({$and:[{“city”:”Jaipur”},{”paymentMode”:”Cash”}]})

**ArrayOperator**

Db.order.find({“OrderItems”:{$all:[“notebook”,”paper”]}}

**Sorting**

db.collectionName.find().sort({Field:Order},{Field:Order},…………)

Order=1(ASC)

Order=-1(DESC)

**Limit in MongoDB**  
db.orders.find(). limit (1)-> Number of records to print

**Relationships in MongoDB**

**One to One**

//Users collection

{

“\_id”:”456efgia”

“username”:”suraj”

“age”:23

}

//Address Collection

{  
\_id:”35fg2i39”

“address”:Jayanagar”

}

//Write directly in user collection

{

“\_id”:”456efgia”

“username”:”suraj”

“age”:23

“address”: [{  
\_id:”35fg2i39”

“address”:Jayanagar”

}]

}

**One to Many RelationShip**

Example

Write just write the ids in the author collection, your books will come in the Author Collection

Before this you should have created the books and author collection priorly

//Author Collection

{

“\_id”:”4563efig”

“authorName”:” suraj”

“books”: [{\_id:”6593efijk”

\_id:”987zyw”

}]

//Result

{

“\_id”:”456e3fig”

“author\_name”:”suraj”

“books”: [{“\_id”:”6593efijk”

“book\_name”:”Chamber of Secrets”

}]

{“\_id”:”9782gw”

book\_name:”Philosopher’s Stone”

}]

**Map Reduce Working**

var map = function () {emit (this.\_id, this.price);}

// Define the reduce function

var reduce = function (Id, Price) {return Array.sum(Price);}

// Execute the Map-Reduce operation on the "books" collection

db.books.mapReduce(map,reduce,{ out: "result" });

Map Function (map):

The map function is defined to emit key-value pairs based on the input documents in the "books" collection.

For each document, it emits the \_id as the key and the price as the value.

Reduce Function (reduce):

The reduce function takes two parameters: Id and Price.

It uses the Array.sum function to calculate the sum of all values in the Price array.

The result is the total sum of prices for a particular \_id.

Map-Reduce Operation (db.books.mapReduce):

The mapReduce method is invoked on the "books" collection.

It takes the map and reduce functions as arguments.

The output of the Map-Reduce operation is stored in a collection named "result" ({ out: "result" }).

var map =function() {emit(this.\_id,this.price)}

var reduce =function (Id,Price){

return Array.sum(Price)};

db.books.mapReduce(map,reduce,{out:”result”});

**Horizontal Scaling:**

Horizontal scaling involves adding more servers to a distributed system to share the load.

In MongoDB, horizontal scaling can be achieved through replica sets and sharding.

**Replica Sets:**

A replica set is a group of MongoDB servers that maintain the same data set.

It provides high availability and failover support.

One node in the replica set is the primary node, and others are secondary nodes.

In the event of a primary node failure, one of the secondaries automatically becomes the new primary.

**Sharding:**

Sharding is a method of distributing data across multiple machines to improve horizontal scalability.

MongoDB uses sharding to divide a large dataset into smaller, more manageable chunks called shards.

Each shard is a separate database server that stores a subset of the data.

The sharded cluster consists of shard servers, mongos routers, and a config server.

**Vertical Scaling:**

Vertical scaling involves adding more resources (CPU, RAM, storage) to a single server to handle increased load.

MongoDB supports vertical scaling by upgrading hardware, such as moving to a server with more powerful processors or additional memory.

Vertical scaling has limits, and once those limits are reached, horizontal scaling is typically a more viable solution.

**Demo Project on MongoDB**

**Dependencies Used:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

**Application Properties file**

server:

port: 8084

spring:

data:

mongodb:

uri: mongodb://localhost:27017/employeedb

MongoRepository will be used instead of Jpa Repository

public interface EmpRepository extends MongoRepository<Employee, Integer>

{

}

**Annotations:**

@Document(collection = "employee"): Indicates that this class is a MongoDB document and specifies the collection name as "employee."

@Id: Marks the field empId as the primary identifier for this document.

**Fields**:

empId: Represents the employee's identifier. It is annotated with @Id, indicating it's the primary key.

empFirstName: Represents the employee's first name.

empLastName: Represents the employee's last name.

email: Represents the employee's email address

@Document(collection = "employee")

public class Employee implements Serializable {

@Id

private Integer empId;

private String empFirstName;

private String empLastName;

private String email;

**EmpController Class**

**Annotations:**

@RestController: Indicates that this class is a Spring MVC controller, and the methods inside it will return the response body directly (rather than relying on a view).

@RequestMapping (path = "/employees"): Specifies the base URL path for all endpoints in this controller.

@OpenAPI30: An annotation from Swagger (OpenAPI 3.0) to generate API documentation.

**Autowired Field:**

@Autowired private EmpService empService;: Injects an instance of the EmpService into the controller. This service likely contains the business logic for handling employee-related operations.

Request Mapping and CRUD Operations:

@PostMapping("/save"): Handles HTTP POST requests to create a new employee.

@GetMapping("/getAll"): Handles HTTP GET requests to retrieve a list of all employees.

@GetMapping("/get/{empId}"): Handles HTTP GET requests to retrieve an employee by ID.

@PutMapping("/update"): Handles HTTP PUT requests to update an existing employee.

@DeleteMapping("/delete/{empId}"): Handles HTTP DELETE requests to delete an employee by ID.

**Method Parameters:**

@RequestBody Employee employee: Used in the createEmployee and updateEmployee methods to deserialize the request body into an Employee object.

@PathVariable Integer empId: Used in the getEmployeeById and deleteEmployee methods to extract the empId from the URL path.

**Exception Handling:**

throws NotFoundException: Indicates that the updateEmployee and deleteEmployee methods may throw a NotFoundException. This exception likely corresponds to a scenario where an employee with the specified ID is not found.

*@RestController*

*@RequestMapping (path = "/employees")*

*@OpenAPI30*

*public class EmpController {*

*@Autowired*

*private EmpService empService;*

*@PostMapping("/save")*

*public Employee createEmployee(@RequestBody Employee employee) {*

*return empService.createEmployee(employee);*

*}*

*@GetMapping("/getAll")*

*public List<Employee> getAllEmployees() {*

*return empService.getAllEmployees();*

*}*

*@GetMapping("/get/{empId}")*

*public Optional<Employee> getEmployeeById(@PathVariable Integer empId) {*

*return empService.getEmployeeById(empId);*

*}*

*@PutMapping("/update")*

*public Employee updateEmployee (@RequestBody Employee updatedEmployee)*

*throws NotFoundException {*

*return empService.updatedEmployee( updatedEmployee);*

*}*

*@DeleteMapping("/delete/{empId}")*

*public void deleteEmployee(@PathVariable Integer empId) throws NotFoundException {*

*empService.deleteEmployee(empId);*

*}*

*}*

**EmpService Class**

**Annotations:**

@Service: Indicates that this class is a Spring service component.

**Autowired Field:**

@Autowired EmpRepository empRepository;: Injects an instance of the EmpRepository into the service. The repository likely extends Spring Data's JpaRepository and provides methods for interacting with the underlying database.

**Methods:**

createEmployee: Saves a new employee to the database using the save method provided by the EmpRepository.

getAllEmployees: Retrieves a list of all employees from the database using the findAll method provided by the EmpRepository.

getEmployeeById: Retrieves an employee by ID from the database using the findById method provided by the EmpRepository. Returns an Optional<Employee> to handle cases where the employee may not exist.

updateEmployee: Updates an existing employee in the database. Checks if the employee with the specified ID exists using existsById before saving the updated employee. Throws a NotFoundException if the employee is not found.

deleteEmployee: Deletes an employee by ID from the database. Checks if the employee with the specified ID exists using existsById before deleting. Throws a NotFoundException if the employee is not found.

**Exception Handling:**

The updatedEmployee and deleteEmployee methods throw a NotFoundException when attempting to update or delete an employee that does not exist. This provides a way to handle situations where an operation is performed on a non-existent employee.

*@Autowired*

*EmpRepository empRepository;*

*public Employee createEmployee(Employee e) {*

*return empRepository.save(e);*

*}*

*public List<Employee> getAllEmployees() {*

*return empRepository.findAll();*

*}*

*public Optional<Employee> getEmployeeById(Integer empId) {*

*return empRepository.findById(empId);*

*}*

*public Employee updatedEmployee(Employee updatedEmployee) throws NotFoundException {*

*if (empRepository.existsById(updatedEmployee.getEmpId())) {*

*return empRepository.save(updatedEmployee);*

*} else {*

*throw new NotFoundException("Employee not found with ID: " + updatedEmployee.getEmpId());*

*}*

*}*

*public void deleteEmployee(Integer empId) throws NotFoundException {*

*if (empRepository.existsById(empId)) {*

*empRepository.deleteById(empId);*

*} else {*

*throw new NotFoundException("Employee not found with ID: " + empId);*

*}*

*}*