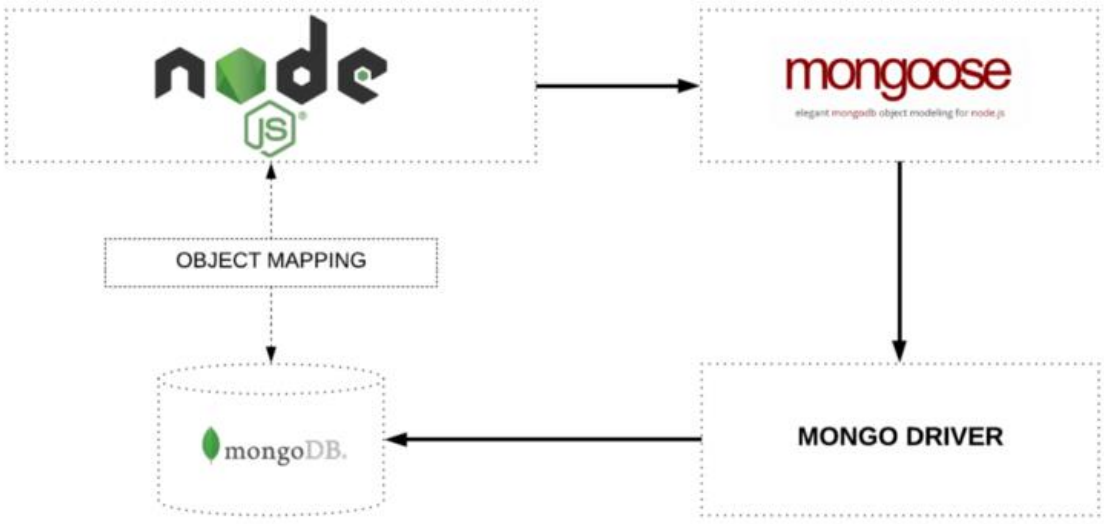
MongoDB -2

Connection with NodeJS

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.



Object Mapping between Node and MongoDB managed via Mongoose

MongoDB is a schema-less NoSQL document database. It means you can store JSON documents in it, and the structure of these documents can vary as it is not enforced like SQL databases. This is one of the advantages of using NoSQL as it speeds up application development and reduces the complexity of deployments.

Mongoose is built on top of the MongoDB driver to provide us with a means to model data easily.

Advantages of mongoose:

1. Validation of the MongoDB database collections can be done very effectively with mongoose.
2. Implementation of predefined structures on Collections is done quickly.
3. Mongoose module provides the abstraction layer for reading and defining a query.

Disadvantages of mongoose:

The main disadvantage of mongoose is that the abstraction comes at the cost of performance as compared to the MongoDB driver. MongoDB driver is around 2x faster than the Mongoose.

Why Mongoose?

By default, MongoDB has a flexible data model. This makes MongoDB databases very easy to alter and update in the future. Mongoose forces a semi-rigid schema from the beginning. With Mongoose, developers must define a Schema and Model.

First be sure you have MongoDB and Node.js installed.

Install Mongoose from the command line using npm:

```
npm install mongoose
```

Step 1: Importing Mongoose and Connecting to MongoDB

The first thing we need to do is include mongoose in our project and open a connection to the test database on our locally running instance of MongoDB.

```
const mg = require('mongoose');
mg.connect("mongodb://127.0.0.1:27017/test").then(()=>{console.log("success")}).catch((err)=>{console.error(err)});
```

- **const mg = require("mongoose"); :** This imports the Mongoose library, which is an Object Data Modeling (ODM) library for MongoDB and Node.js.
- **mg.connect("mongodb://127.0.0.1:27017/lju"):** This line establishes a connection to a MongoDB database named **lju** running on the local machine (localhost) at the default port 27017.
- **.then(...):** If the connection is successful, it logs "success" to the console.
- **.catch(...):** If there is an error during connection, it logs the error to the console.

Step 2: Defining a Schema

- ✓ **const mySchema = new mg.Schema({ ... });:** This defines a new schema for a collection. A schema defines the structure of the documents within a collection.

What is a schema?

A schema defines the structure of your collection documents. A Mongoose schema maps directly to a MongoDB collection.

```
const myschema=new mg.Schema({
   name:{type:String,required:true}, //name must be given else error
   surname:String,
   age:Number,
   active:Boolean,
   date:{type:Date,default: new Date().toLocaleDateString()
})
```

- The schema includes the following fields:
  - **name:** A required string field.
  - **Surname**: An optional string field.
  - **age:** An optional number field.

  - **active:** An optional boolean field.
  - **date:** A date field with a default value set to the current date, formatted as a locale date string.
- The default value for the date field is set using new Date().toLocaleDateString(), which will set the date as a string formatted according to the local date format.

With schemas, we define each field and its data type. **Permitted types are:** String,Number,Date,Buffer,Boolean,Mixed,ObjectId,Array,Decimal128,Map etc.

Step 3: Creating a Model

What is a model?

Models take your schema and apply it to each document in its collection.

Models are responsible for all document interactions like creating, reading, updating, and deleting (CRUD).

An important note: the first argument passed to the model should be the singular form of your collection name. Mongoose automatically changes this to the plural form, transforms it to lowercase, and uses that for the database collection name.

```
const person=new mg.model("person",mySchema)
```

it will automatically converted to "people" by mongoose. To avoid Auto plural use below command.

```
mg.pluralize(null) // to add collection as it is we have mentioned before Creating model
```

Step 4: Creating and Saving a Document

Type-1 Insert a new data one by one using .save() method Back in the file.

```
const persondata=new person({
 name:"DDD",
surname:"PQR2",
age:2,
active:true
})
persondata.save()

const persondata2=new person({
name:"ABC2",
surname:"PQR2",
age:2,
active:true
})
persondata2.save()
```

we create a new object and then use the save() method to insert it into our MongoDB database.

Type 2: Using Async..await

- **const createDoc = async () => { ... };:** This defines an asynchronous function createDoc to create and save a document in the MongoDB collection.
- **const personData = new person({ ... });:** This creates a new instance of the person model with the specified data:
  - name: "test"
  - Surname: "XYZ"
  - age: 3
  - active: true
- **const result = await personData.save();:** This saves the document to the database and waits for the operation to complete.
- **console.log(result);:** If the document is successfully saved, it logs the resulting document to the console.
- **catch (err) { ... }:** If an error occurs during the save operation, it logs the error to the console.

Await and Async

If you are using async functions, you can use the await operator on a Promise to pause further execution until the Promise reaches either the Fulfilled or Rejected state and returns. Since the await operator waits for the resolution of the Promise, you can use it in place of Promise chaining to sequentially execute your logic.

The async and await keywords in JavaScript are used to handle asynchronous operations in a more readable and straightforward way compared to traditional methods like callbacks and Promises. Here's an explanation of their use and benefits:

async Keyword

The async keyword is used to define an asynchronous function. When a function is declared as async, it implicitly returns a Promise. This means you can use the await keyword inside this function to pause its execution until the awaited Promise is resolved or rejected.

await Keyword

The await keyword can only be used inside an async function. It pauses the execution of the async function and waits for the resolution (or rejection) of a Promise. Once the Promise is resolved, the function resumes execution, and the resolved value of the Promise is returned. If the Promise is rejected, the await expression throws the rejected value.

Benefits of async and await

1. **Readability**: The syntax of async and await makes asynchronous code look more like synchronous code, which is easier to read and understand.
2. **Error Handling**: Handling errors with async/await is more straightforward using try/catch blocks compared to handling errors with Promises.
3. **Avoiding Callback Hell**: async/await helps to avoid deeply nested callbacks, making the code cleaner and more maintainable.

Example 1: Add name, Surname,Age,Active and date field in persons collection inside Test database. (Select an appropriate Type as per need)
// Type:1 To insert one by one data
// Type:2 Raise exception and catch it, if any problem occurs. Print a proper message for error handling.
// Type:3 To insert Multiple data at once

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test").then(()=>{console.log("success")}).catch((err)
=>{console.error(err)});
//mg.pluralize(null)
const mySchema=new mg.Schema({
      name:{type:String, required:true},
      Surname:String,
      age:Number,
      date:{type:Date, default:new Date()}
//This adds date field with all dates set in ISO format.ISO format is yyyy-mm-dd included with time 5:35:35. The time has zero time zone, so we have to add 5:30, because India has GMT+5:30 time zone.

   });
const person=new mg.model("person",mySchema)
// Type:1 To insert one by one data
const personData=new person({ name:"ABC",  Surname:"Patel",   age:30})
personData.save() // also returns promise

const personData1=new person({ name:"XYZ",  Surname:"Patel",   age:30})
personData1.save() // also returns promise

//OR - Type:2
const createdoc=async()=>
{
try
   {
      const persondata=new person({name:" ABC ",surname:" Patel ",age:30})
      const result=await persondata.save();
      console.log(result)
   }
catch(err){console.log("There is error")}
}
createdoc()
```

```
//OR -  Type:3 To insert Multiple data at once
const createDoc=async()=>
   {
      try{
      const personData=new person({ name:"ABC",  Surname:"Patel",   age:30})
      const personData1=new person ({ name:"XYZ",  Surname:"Patel",   age:30})
      const result= await person.insertMany   ([personData , personData1])
      console.log(result)
      }
         catch(err){
            console.log("problem")}
   }
createDoc();
```

Summary

- ✓ The code imports Mongoose and connects to a local MongoDB instance.
- ✓ It defines a schema for the person collection with fields for name, Surname, age, active, and date.
- ✓ It creates a model for the person collection based on the schema.
- ✓ It defines an asynchronous function to create a new document and save it to the database.
- ✓ If the document is successfully saved, it logs the document to the console; otherwise, it logs an error.

In Mongoose and MongoDB, __v is a special field that stands for **version**. It is used to track the version of a document within a collection. Here's a detailed explanation:

Purpose of __v

id: ObjectId('66c1209c8ba3569ea4cd15f7')
name: "john doe"
age: 25
email: "john.doe@example.com"
gender: "MALE"
__v: 0

Version Tracking:

1. The __v field is automatically managed by Mongoose to handle concurrency control. Each time a document is updated, its version number (__v) is incremented.
2. This helps in managing optimistic concurrency control. If two processes attempt to update the same document simultaneously, Mongoose can use the version number to detect conflicts and prevent data loss.

Example2:
Write a node.js script to enter more than one document to the collection after establishing a connection using mongoose. Raise exception and catch it, if any problem occurs. Print a proper message for error handling. Else find some records.

Insert by creating an array of objects

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test").then(()=>{console.log("success")}
).catch((err)=>{console.error(err)});
//mg.pluralize(null)
const mySchema=new mg.Schema({
      name:{type:String, required:true },
      Surname:String,  age:Number,   active:Boolean,  date:{ type:Date, default:new Date()}
   });
const person=new mg.model("person",mySchema)
const createDoc=async()=> {
      try{
         const personData1=[
         {name:"test", Surname:"test1", age:33, active:true},
         {name:"hi", Surname:"hi1", age:30, active:true},
         {name:"hello", Surname:"hello1", age:37, active:true},
         {name:"hello",Surname:"hello11", age:37, active:true}]

          const result= await person.insertMany(personData1)
         console.log(result)
      const result1= await person.find({name:"abc"},{date:1,_id:0})
      const result2= await person.find({name:"hi"}).limit(1)
      const result3= await person.find({name:"hello1"}).select({name:1}).limit(1)
      const result4= await person.findOne({name:"abc"}).select({Surname:1})
      const result5= await person.find({age:{$gt:28}},{name:1,_id:0})
      const result6= await person.find({name:{$in:["abc","pqr"]}})
      const result7= await person.find({age:{$gt:0,$lt:28}})
      const result8=await person.deleteMany({Surname:"hello1"})
      const result9=await person.find({age:{$lte:28}}).sort({name:-1}).count()
         console.log(result1)
         console.log(result2)
         console.log(result3)
         console.log(result4)
         console.log(result5)
         console.log(result6)
         console.log(result7)
         console.log(result8)
         console.log(result9)
      }
         catch(err) { console.log("problem"); }
   }
createDoc();
```

Update / Delete

The methods updateOne, findByIdAndUpdate, and findByIdAndDelete in Mongoose are used for different operations on MongoDB documents. Below is an explanation of each, their differences, and when to use them.

1. updateOne

- Updates a single document that matches a given query.

```
Model.updateOne(filter, update, options, callback)
```

- **Parameters:**
  - filter: An object that specifies the criteria to find the document to update.
  - update: An object containing the fields to update.
  - options (optional): An object containing options like:
    - upsert: If true, creates a new document if no document matches the filter. Default is false.
    - runValidators: If true, runs schema validators on the update operation. Useful for maintaining data integrity.
  - callback (optional): A function to execute once the operation completes.
- **Use Cases:**
  - When you want to update a specific document based on a custom query (not just _id).
  - When you need to ensure that only one document is updated, even if multiple documents match the query.

2. findByIdAndUpdate

- Finds a document by its _id and updates it.

```
Model.findByIdAndUpdate(id, update, options, callback)
```

- **Parameters:**
  - id: The _id of the document to find and update.
  - update: An object containing the fields to update.
  - options (optional): An object containing options like:
    - new: If true, returns the updated document instead of the original. Default is false.
    - runValidators: If true, runs schema validators on the update operation.
    - upsert: If true, creates a new document if no document matches the _id. Default is false.
  - callback (optional): A function to execute once the operation completes.
- **Use Cases:**
  - When you need to update a document by its unique _id.
  - When you want to retrieve the updated document immediately after the update.

3. findByIdAndDelete

- Finds a document by its _id and deletes it.

```
Model.findByIdAndDelete(id, options, callback)
```

- **Parameters:**
  - id: The _id of the document to find and delete.
  - options (optional): You can pass an options object, though it's rarely needed for deletion.

- callback (optional): A function to execute once the operation completes.

- **Use Cases:**
  - When you need to delete a document by its _id.
  - When you want to retrieve the document being deleted for confirmation.
  -

Summary

- **updateOne:** Updates the first document matching a filter. Useful for more complex queries.
- **findByIdAndUpdate:** Finds a document by _id and updates it. Use this when you know the _id of the document you want to update.
- **findByIdAndDelete:** Finds a document by _id and deletes it. Use this when you want to remove a document and potentially get the deleted document back.

Each method has its own specific use cases and should be chosen based on the operation you need to perform.

Example: create database connection and insert multiple document(record) in collection(table) and **update Document(record) by ID.**

Using Update function

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema({
      name:{ type:String, required:true},
      Surname:String,
      age:Number,
      active:Boolean,
      date:{ type:Date, default:new Date()}
   });
const person=new mg.model("person",mySchema)
 const updateDoc=async(i) =>{
     const result=await person.updateOne({_id:i},
     {
       $set:{age:37}
      })
    console.log(result)
 }
 updateDoc("64a7cd53376fc04b3c2637bd");
```

Using findByIdAndUpdate function

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
   {
      name:{
         type:String,
         required:true
      },
      Surname:String,
      age:Number,
      active:Boolean,
      date:{
         type:Date,
         default:new Date()
      }
   }
);
const person=new mg.model("person",mySchema)
 const updateDoc=async(_id) =>
{
     const result=await person.findByIdAndUpdate({_id},
          {
             $set:{age:27}
          },
           {new:true}
      )
    console.log(result)
 }
 updateDoc("64a7cd53376fc04b3c2637bd");
```

Example: create database connection and insert multiple document(record) in collection(table) and **delete Document(record) by ID.**

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/test")
.then(()=>{console.log("success")})
.catch((err)=>{console.error(err)});
mg.pluralize(null)
const mySchema=new mg.Schema(
  {
       name:{
            type:String,
           required:true
      },
       Surname:String,
        age:Number,
       active:Boolean,
       date:{
            type:Date,
            default:new Date()
        }
    }
 );
 const person=new mg.model("person",mySchema)
const deleteDoc=async(_id) =>
{
       const result=await person.deleteOne({_id})
                                OR
       const result=await person.findByIdAndDelete({_id})

    console.log(result)
}
deleteDoc("64a7cd53376fc04b3c2637bd");
```

Mongoose Schema validation

Mongoose Schema validation is a powerful feature that allows you to define rules for your data and ensure that documents saved to the MongoDB database meet these requirements. Validation is performed before saving a document and can help maintain data integrity and consistency.

Schema validation lets you create validation rules for your fields, such as allowed data types and value ranges.

MongoDB uses a flexible schema model, which means that documents in a collection do not need to have the same fields or data types by default.

Schema validation is most useful for an established application where you have a good sense of how to organize your data.

When MongoDB Checks Validation

When you create a new collection with schema validation, MongoDB checks validation during updates and inserts in that collection.

When you add validation to an existing, non-empty collection:

- Newly inserted documents are checked for validation.
- Documents already existing in your collection are not checked for validation until they are modified.

What Happens When a Document Fails Validation

By default, when an insert or update operation would result in an invalid document, MongoDB rejects the operation and does not write the document to the collection.

1. **Basic Built-in validators:**

Mongoose provides several built-in validators for common data validation tasks:

1. **required**: Ensures that a field is present.
2. **min and max**: For number fields, they set minimum and maximum values.
3. **enum**: Ensures that the value of a field is one of a specified set of values.
4. **match**: For string fields, it ensures that the value matches a specified regular expression.
5. **minlength and maxlength**: For string fields, they ensure the length is within a certain range.
6. **trim:** Automatically removes leading and trailing whitespace from a string.
7. **uppercase**: Converts the string to uppercase before saving it to the database.
8. **lowercase**: Converts the string to lowercase before saving it to the database.
9. **default:** Assigns a default value to a field if no value is provided when the document is created.
10. **validate:** Allows for custom validation logic using a function. The function returns true if the value is valid and false otherwise. You can also return a promise for asynchronous validation.

2.**Custom validation:** We can provide custom validation using **validate(value)** inbuild function.

3**. Validator Module**: Using Validator package by using npm i validator

The Validator module is popular for validation. Validation is necessary to check whether the data is correct or not, so this module is easy to use and validates data quickly and easily.

Feature of validator module:

- It is easy to get started and easy to use.
- It is a widely used and popular module for validation.
- Simple functions for validation like isEmail(), isEmpty(), etc.

Example

Write a node.js script to define a schema having fields like name, age, gender,email.

Apply following validations:

(1) Name field must remove leading/trailing spaces,minimum and maximum length should be 3 & 10 respectively, and name should be stored in lowercase

(2) Age must accept value greater than 0.

(3) Perform Email ID validation on Email field.

(4) Gender must accept values in uppercase only and allowed values are “MALE” & “FEMALE” only.

```
const mg=require("mongoose")
const v=require("validator")
mg.connect("mongodb://127.0.0.1:27017/lju").then(()=>{console.log("success")}).catch((err)=
>{console.error(err)});
//mg.pluralize(null)
const mySchema=new mg.Schema( {
      name:{  type:String,  required:true,  lowercase:true,  trim:true,
              minlength:[3,"Min length must be 3"], maxlength:[7,"max length must be 7"],
              //  enum: {values: ['Coffee', 'Tea'], message: '{VALUE} is not supported'}
                  },
     age:{   type:Number,
         validate(v1){
            if(v1<=0)
            {throw new Error("Number must be positive")  }  }  },
      email:{ type:String,
               validate(val){
                 if(!v.isEmail(val))
            { throw new Error("Enter valid email_id") }} },
   gender: { type:String,
         uppercase: true,
         enum:[‘MALE’,’FEMALE’]}
  });
const person=new mg.model("person",mySchema)
const personData=new person({name:"def",age:10,email:abc@gmail.com, gender:”MALE”} )
      personData.save();
```

Connectivity of MongoDB with ExpressJS

Install express if not already installed using command. **npm install express**

Task: Create a form having username and password and insert data entered by user in collection named "data1" in mongoDB.

In task.js file

```
var expr=require("express")
var app=expr()
const mg=require("mongoose")

mg.connect("mongodb://127.0.0.1:27017/login")
.then(()=>{console.log("Successful")})
.catch((err)=>{console.error(err)})

mg.pluralize(null)
const myschema=new mg.Schema({
      uname:{type:String, required:true},
      password: {type:String, required:true}
//{versionKey: false}- TO nullify version key on each doc created by mongoose when 1st created
})
const person =new mg.model("data1", myschema)

app.use(expr.static(__dirname,{index:"form.html"}))
//Or app.get("/",(req,res)=>{ res.sendFile(__dirname+"/form.html") })

app.get("/process_get",(req,res)=>{
     const personData=new person({
     uname:req.query.uname,
     password:req.query.pwd})
personData.save()
res.send("Record inserted")
})
app.listen(3000)
```

form.html

```
<html>
   <form action="/process_get" method="get">
      Username: <input type="text" name="uname"/>
   </br> Password: <input type="password" name="pwd"/>
   </br> <input type="submit"/>
   </form>
</html>
```

Connectivity of MongoDB with React

Set Up a MongoDB Database

- MongoDB is the most popular NoSQL database. It is an open-source database that stores data in JSON-like documents (tables) inside collections (databases).
- Open the MongoDB Compass app. Then, click the **New Connection** button to create a connection with the MongoDB server running locally.
- If you do not have access to the MongoDB Compass GUI tool, you can use the MongoDB shell tool to create a database and the collection.
- Provide the connection URI and the name of the connection, then hit **Save & Connect**.
- Lastly, click on Create Database button, fill in the database name, and provide a collection name for a demo collection.

Folder structure:

React_connection (Main folder)
--Backend (folder for backend) (install packages express,cors,mongoose,body-parser,bcrypt)
---Server.js (backend file)
--Frontend (folder for frontend) (create react app in this folder) (install axios)
---myapp
public
src
---Signup.js (frontend file)
---Login.js (frontend file)

Create a React Client

```
npx create-react-app my-app
cd my-app
npm start
```

Next, install Axios. This package will enable you to send HTTP requests to your backend Express.js server to store data in your MongoDB database.

```
npm install axios
```

Create a Demo Form to Collect User Data

Signup.js

```
import React, { useState } from 'react';
import axios from 'axios';

function Signup() {
  const [username, setUsername] = useState('');

  const handleSignup = async (e) => {
    e.preventDefault();

    try {
      await axios.post('http://localhost:5000/signup', {username});
     alert('User signed up successfully.'+username);
     document.getElementById("test").innerHTML=username
      setUsername('');
     }
 catch (error) {
      console.error('Error signing up:', error);
     alert('An error occurred.');
    }
  };
  return (
    <div>
      <h1>Sign Up</h1>
      <form onSubmit={handleSignup} method ='post'>
      Name: <input type="text" placeholder="Username" value={username}
         onChange={(e) => setUsername(e.target.value)} />
        <button type="submit">Sign Up</button>
<h1 id="test"></h1>
      </form>
    </div>  )}

export default Signup;
```

App.js

```
import Signup from "./Signup"
function App(){
return(
<>
 <Signup/>
</>
  )}
export default App
```

To run use → npm start

- Declare one state a name to hold the user data collected from the input field using the useState hook.
- The **onChange** method of each input field runs a callback that uses the state methods to capture and store data the user submits via the form.
- To submit the data to the backend server, the onSubmit handler function uses the **Axios.post** method to submit the data passed from the states as an object to the backend API endpoint.

Axios

Fetching User Data: If you want to retrieve user information from the server:

```
axios.get('http://localhost:3000/users')
  .then(response => {
    // Process the response data
  });
```

Creating a New User: If you want to create a new user by sending data to the server:

```
axios.post('http://localhost:3000/signup', { username: 'newUser' })
  .then(response => {
    // Handle success
  });
```

Create an Express js Backend

An Express backend acts as middleware between your React client and the MongoDB database. From the server, you can define your data schemas and establish the connection between the client and the database.

Create an Express web server and install these packages:

```
npm install mongoose
npm install cors
```

The cors middleware

- It is used to enable Cross-Origin Resource Sharing (CORS). CORS is a security feature implemented by web browsers that restricts how resources on a web page can be requested from a different domain than the one that served the web page.
- **Why CORS is Important:**
- When you're developing a frontend (e.g., React) and backend (e.g., Express) separately, they often run on different domains or ports during development (e.g., React on localhost:3000 and Express on localhost:5000). This difference can trigger the browser's same-origin policy, which blocks requests from the frontend to the backend.
- CORS is necessary to allow the frontend to communicate with the backend. By using the cors middleware, you tell the Express server to include specific headers in its responses, enabling the frontend to bypass the same-origin policy.

- **How CORS is Used in the Code:**
- app.use(cors());
- This line adds the cors middleware with the default settings, which allows requests from any origin (i.e., all domains). This is particularly useful during development, but in a production environment, you might want to restrict it to specific domains.

express.json():

- This middleware is added to ensure that JSON payloads are correctly parsed and available in req.body.
- **Body Parsing**: The express.urlencoded({ extended: false }) middleware only parses URL-encoded bodies (typically from forms). Since your React application likely sends data as JSON, the server does not understand it, leading to req.body being empty.
- Add express.json() to parse incoming JSON requests.

server.js

```
const express = require('express');
const mg = require('mongoose');
const cors = require('cors');
const app = express();

app.use(cors());
app.use(express.json());
mg.connect('mongodb://127.0.0.1:27017/User') .then(()=>{console.log("Connection
Success")})
const UserSchema = new mg.Schema({  username:String  });

const User = new mg.model('User', UserSchema);

app.post('/signup', async (req, res) => {
  try {
    const { username} = req.body;
//console.log(“Username is” + req.body.username)

    const newUser = new User({ username});
    await newUser.save();
    res.send();
// or res.json({message:’Username inserted Successfully’})
  } catch (error) {
    res.send(error);
//or res.json({message:’Error in inserted Record’})
  }
});
app.listen(5000);
```

hit: node server.js

Indexing in MongoDB

MongoDB uses indexing in order to make the query processing more efficient. If there is no indexing, then the MongoDB must scan every document in the collection and retrieve only those documents that match the query. Indexes are special data structures that stores some information related to the documents such that it becomes easy for MongoDB to find the right data file. The indexes are order by the value of the field specified in the index.

When not to create indexing:

- When collection is small
- When collection is updated frequently
- When queries are complex
- When collection is large and multiple indexes are applied.

If your application is repeatedly running queries on the same fields, you can create an index on those fields to improve performance. For example, consider the following scenarios:

Scenario	Index Type
A human resources department often needs to look up employees by employee ID. You can create an index on the employee ID field to improve query performance.	**Single Field Index**
A store manager often needs to look up inventory items by name and quantity to determine which items are low stock. You can create a single index on both the item and quantity fields to improve query performance.	**Compound Index**

- Indexes are special data structures that store a small portion of the collection's data set in an easy-to-traverse form.
- MongoDB indexes use a B-tree data structure.
- The index stores the value of a specific field or set of fields, ordered by the value of the field.

Default Index
MongoDB creates a unique index on the _id field during the creation of a collection. The _id index prevents clients from inserting two documents with the same value for the _id field. You cannot drop this index.

Index Names
The default name for an index is the combination of the **index keys** and each key's **direction** in the index **(1 or -1)** with **underscores** as a separator.

For example, an index created on { item : 1, quantity: -1 } has the name item_1_quantity_-1.

explain() method

Provides information on the query plan for the db.collection.find() method.

Syntax: db.collection.find().explain(**"executionStats"**)

The following example runs cursor.explain() in "executionStats" verbosity mode to return the query planning and execution information for the specified db.collection.find() operation:

```
db.students.find({age:{$gt:15}}).explain("executionStats")
```

Explain(executionStats) is a method that you can apply to simple queries or to cursors to investigate the query execution plan. The execution plan is how MongoDB resolves a query. Looking at all the information returned by explain():

- **totalDocsExamined**: how many documents were examined
- **nReturned**:how many documents were returned
- **stage:** Inside the winningPlan -> queryPlan -> stage is defined. It is used to defined the stage of input scanning.
    - COLLSCAN for a collection scan
    - IXSCAN for scanning index keys

Creating an Index

MongoDB provides a method called createIndex() that allows user to create an index.

Syntax: db.**collection_name**.createIndex({KEY:1})

MongoDB only creates the index if an index of the same specification does not exist.
The key determines the field on the basis of which you want to create an index and 1 (or -1) determines the order in which these indexes will be arranged

1 for ascending order and -1 for descending order

1. **Create an Index on a Single Field**

We can create an index on any one of the fields of collection.

Example:

```
db.table1.createIndex({age:1})
```

This will create an index named "age_1".

***Read documents using Index:**

Syntax: db.collection_name.find().explain("executionStats")

2. Compound indexing

Compound indexes are indexes that contain references to multiple fields. Compound indexes improve performance for queries on exactly the fields in the index or fields in the index prefix. To create a compound index, use the db.collection.createIndex() method:

Syntax:

db.<collection>.createIndex({<field1>:<sortOrder>,<field2>:<sortOrder>,...,<fieldN>: <sortOrder>})

Example:

```
db.table1.createIndex({age:1,name:-1})
```

An index created on { age : 1, name: -1 } has the name age_1_name_-1.

In this example:

- The index on age is ascending (1).
- The index on name is descending (-1).

The created index supports queries that applies on:

- Both age and name fields.
- Only the age field, because age is a prefix of the compound index.

For example, the index supports these queries: (Perform IXSCAN)

```
db.students.find( { name: "Abc", age: 36 } )
db.students.find( { age: 26 } )
```

The index does not support queries on only the name field, because name is not part of the index prefix. For example, the index does not support this query:

```
db.students.find( { name: "Abc" } ) →perform CoLLSCAN
```

- **getIndexes()**

  To confirm that the index was created, use below command:

  ```
  db.collection.getIndexes()
  ```

  Output:

  [{ v: 2, key: { _id: 1 }, name: '_id_' },{ v: 2, key: { age: 1 }, name: 'age_1' }]

- **dropIndex()/dropIndexes()**

  dropIndex() : Drops or removes the specified index from a collection.

  ```
  db.collection.dropIndex(index)
  ```

  dropIndexes() : It is used to drop all indexes except the _id index from a collection.

  ```
  db.collection.dropIndexes()
  ```

Drop a specified index from a collection. To specify the index, you can pass the method either:

- **The index specification document**
  db.collection.dropIndexes({ a: 1, b: 1 })

- **The index name:**
  db.collection.dropIndexes("a_1_b_1")

- **Drop specified indexes from a collection. To specify multiple indexes to drop, pass the method an array of index names:**
  db.collection.dropIndexes(["a_1_b_1", "a_1", "a_1_id_-1"])

If the array of index names includes a non-existent index, the method errors without dropping any of the specified indexes

To get the names of the indexes, use the db.collection.getIndexes() method.

Example:

Let's understand the concept with following example.

Suppose, we have a collection named student and we want to apply query to find students who **have age greater than 12**.

Student Collection

_id	Name	Age
1	ABC	10
2	XYZ	12
3	ABC	15
4	PQR	13
5	XYZ	15
6	ABC	8

- **Find students without creating an index.**

db.student.find({age:{$gt:12}}).explain("executionStats")
It will examine 6 documents and return 3 documents by performing COLLSCAN.
totalDocsExamined: 6
nReturned:3
stage: COLLSCAN

- **Create an index on the age field to improve performance for those queries:**

db.student.createIndex({ age: 1 })
index name: age_1

- **Now, find the students with age greater than 12**

db.student.find({age:{$gt:12}}).explain("executionStats")
It will examine 3 documents and return 3 documents by performing IXSCAN.
totalDocsExamined: 3
nReturned: 3
stage: IXSCAN

Compound Indexing

- **Create an index on the age and name fields to improve performance.(Compound Indexing)**

Note: Before creating any other indexing on same fields. Drop the created indexing

db.student.createIndex({ age: 1,name:-1 })
index name: age_1_name_-1

_id	name	age
1	ABC	8
2	ABC	10
3	XYZ	12
4	PQR	13
5	XYZ	15
6	ABC	15

- So here on name field indexing is applied based on age field.
- In this example we have two students with age 15. For age field values 15 the name field values are arranged in descending order.
- name field indexing is depending on the age field. So if we apply query on name field then it will perform the collscan.

Now, consider below three scenarios

1. Display student with name “ABC” and age 15

db.student.find({age:15,name:”ABC”}).explain("executionStats")
It will examine 1 documents and return 1 documents by performing IXSCAN.
totalDocsExamined: 1
nReturned: 1
stage: IXSCAN

2. Display students whose age is greater than 12

db.student.find({age:{$gt:12}).explain("executionStats")
It will examine 3 documents and return 3 documents by performing IXSCAN.
totalDocsExamined: 3
nReturned: 3
stage: IXSCAN

3. Display students whose name is “ABC”

db.student.find({name:”ABC”}).explain("executionStats")
It will examine 6 documents and return 3 documents by performing COLLSCAN.
totalDocsExamined: 6
nReturned: 3
stage: COLLSCAN

Partial Indexing

The partial index functionality allows users to create indexes that match a certain filter condition. Partial indexes use the partialFilterExpression option to specify the filter condition. The partialFilterExpression option accepts a document that specifies the filter condition using:

- equality expressions (i.e. field: value or using the $eq operator),
- $exists: true expression,
- $gt, $gte, $lt, $lte expressions,
- $type expressions,
- $and operator,
- $or operator,
- $in operator

Example:

Consider student collection with 6 documents

_id	name	age
1	ABC	16
2	ABC	10
3	XYZ	12
4	PQR	13
5	XYZ	15
6	ABC	15

- **Create index on age where age is greater than 15**

```
db.student.createIndex({age:1},{partialFilterExpression:{age:{$gt:15}}})
```

- **Display student/s whose age is 16.**

```
db.student.find({age:16}).explain("executionStats")
```

Output: stage: 'IXSCAN',
nReturned: 1,
docsExamined: 1

We have applied index on age field where age is greater than 15. we are looking for age 16 which is greater than 15. Then here it will perform the IXSCAN.

- **Display student/s whose age is 13.**

```
db.student.find({age:13}).explain("executionStats")
```

Output: stage: 'COLLSCAN',
nReturned: 1,
docsExamined: 6

Suppose we have only one document available with age value 13.

We have applied index on age field where age is greater than 15. we are looking for age 13 which is less than 15. Then here it will perform the COLLSCAN.

Winning plan:

If we have applied same field indexing on multiple times, then searching can be done using one type of indexing only. In this scenario, it races with one indexing strategy and only one indexing strategy wins, the other strategy is stored in loosing plan (rejected plan).

The winning plan is stored in cache till 1000 write operations and next time it does not race with matching index. So, this index is stored in winning plan and the other index is stored in rejected plan.

Let's say we have already one index on age and one more index apply on age like:

Code: db.DATA.createIndex({age:1,name:1})

```
db.DATA.find({age:19}).explain("executionStats")
```

so, two indexes have been created on age field, then it selects only one index for searching called inside winning plan, rest of the index is called as rejected plan.

Example:

Consider a collection student having documents like this:

```
[
{_id:123433,name: "DDD",age:32},
{_id:123434,name: "BBB",age:20},
{_id:123435,name: "AAA",age:10},
]
```

Do as directed:

(1) Create an index & fire a command to retrieve a document having age>15 and name is "BBB". Stats must return values nReturned=1, docExamined=1, stage="IXSCAN". Perform required indexing.

(2) Create an index on subset of a collection having age>30. Also write a command to get a stats "IXSCAN" for age>30.

Solution:

1) db.student.createIndex({age:1,name:1})
   db.student.find({age:{$gt:15},name:'DDD'}).explain('executionStats')

2) db.student.createIndex({age:1},{partialFilterExpression:{age:{$gt:30}}})
   db.student.find({age:{$gt:30}}).explain('executionStats')

MongoDb Replication

Replication is the process of synchronizing data across multiple servers. It provides redundancy and increase data availability with multiple copies of data on diff. DB servers.

Replication protect a DB from loss of a single server.It also allows you to recover from Hardware failure and service interruption.

Redundancy and Data Availability

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centers can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

Replication Key Features :

- Replica sets are the clusters of N different nodes that maintain the same copy of the data set.
- The primary server receives all write operations and record all the changes to the data/
- The secondary members then copy and apply these changes in an asynchronous process.
- All the secondary nodes are connected with the primary nodes. there is one heartbeat signal from the primary nodes. If the primary server goes down an eligible secondary will hold the new primary.

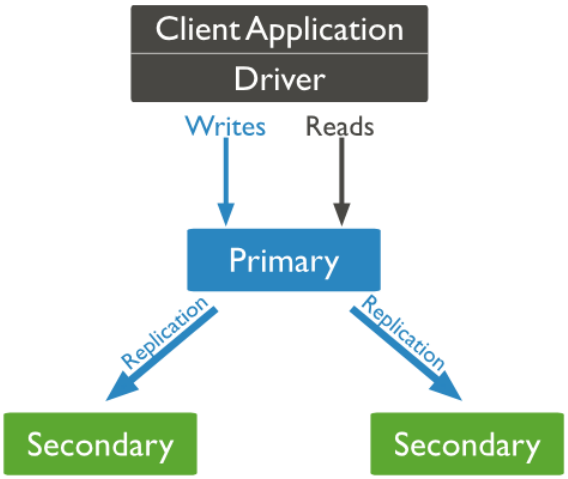
Why Replication?

- High Availability of data disasters recovery
- No downtime for maintenance (like backups index rebuilds and compaction)
- Read Scaling (Extra copies to read from)

How replication works

Replica set is a group of two or more nodes.

In a replica set, one node is Primary node and remaining nodes are secondary.



- Data of primary server is copied to secondary. This duplication is done asynchronously.

Sharding

Sharding is a method for allocating data across multiple machines. MongoDB used sharding to help deployment with very big data sets and large throughput the operation. By sharding, you combine more devices to carry data extension and the needs of read and write operations Sharding solve the the problem of horizontal scaling.

The vertical scaling approach

The vertical scaling approach, sometimes referred to as "scaling up," focuses on adding more resources or more processing power to a single machine. These additions may include CPU and RAM resources upgrades which will increase the processing speed of a single server or increase the storage capacity of a single machine to address increasing data requirements.

The horizontal scaling approach

The horizontal scaling approach, sometimes referred to as "scaling out," entails adding more machines to further distribute the load of the database and increase overall storage and/or processing power. There are two common ways to perform horizontal scaling — they include sharding, which increases the overall capacity of the system, and replication, which increases the availability and reliability of the system.

Automatic Failover

When a primary does not communicate with the other members of the set for more than the configured electionTimeoutMillis period (10 seconds by default), an eligible secondary calls for an election to nominate itself as the new primary. The cluster attempts to complete the election of a new primary and resume normal operations.

In a **three member replica set with two secondaries**, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary.

Steps for Replication

Steps	Command
1	Prepare new folder "data" in D drive. Inside that another folder named db. Inside that make two sub folder db1 and db2.
2	Open **cmd no.1** and hit below command (Primary server) mongod --port 27018 --dbpath "D:\data\db\db1" --replSet rs1
3	Open **cmd no.2** and hit below command (Secondary server) mongod --port 27019 --dbpath "D:\data\db\db2" --replSet rs1
4	Open **cmd no.3** to check replica created or not by using below command mongosh --port 27018 Provides an interface to perform query for port number 27018. (if it opens "test>" then created successfully)
5	To initiate replica set **cmd no.3** The following example initiates a new replica set with two members. rs.initiate({ _id:"rs1", members:[{_id:0, host:"127.0.0.1:27018"}, {_id:1, host:"127.0.0.1:27019"}] })
6	Cmd no.3 : **rs.status() –** Returns the replica set status from the point of view of the member where the method is run. Shows that 127.0.0.1:27020 is the primary server Shows that 127.0.0.1:27021 is the Secondary server
7	Cmd no 3: rs1 [direct: primary] test> show dbs 3 Dbs named admin,config and local. On New server port :27018

8	Cmd no 3: Switch db by using command rs1 [direct: primary] test> use pspdata **Insert record** rs1 [direct: primary] pspdata> db.people.insertMany([{ name:”ABC”,age:29,dept:”IT”},{ name:”PQR”,age:29,dept:”IT”}]) This will inset record also in 27019. Open mondoDB compass change port and check.
9	Data automatically inserted secondary server but still not able to read as permission required for that. **Open cmd 4:** mongosh --port 27019 use pspdata db.getMongo().setReadPref(“primaryPreferred”) To specify Read Preference Mode The following operation sets the read preference mode to target the read to a primary member. This implicitly allows reads from primary. **Read Preference Mode** • **primary** Default mode. All operations read from the current replica set primary. • **primaryPreferred** In most situations, operations read from the primary but if it is unavailable, operations read from secondary members. • **secondary** All operations read from the secondary members of the replica set. • **secondaryPreferred** Operations typically read data from secondary members of the replica set. If the replica set has only one single primary member and no other members, operations read data from the primary member. • **nearest** Operations read from a random eligible replica set member, irrespective of whether that member is a primary or secondary, based on a specified latency threshold.
10	Fire find command cmd 4: db.people.find()
11	Try to enter data in secondary server db.people.insertOne({name:”a1”}). It will not allow to enter record. Shows error message : not Primary

Miscellaneous Task

Task- 1

You are developing a MongoDB-based application using Mongoose. You need to define a userSchema that includes various validation rules to ensure data integrity and consistency. Define a Mongoose schema called userSchema with the following fields and validation requirements:

- username:
  - Required and must be between 4 and 20 characters long.
  - Must start with letters and end with digits.
  - Should be trimmed of any leading or trailing spaces.
  - Should be converted to uppercase before saving.
- email:
  - Required, must be unique across the collection.
  - Must follow the standard email format.
- age:
  - Must be a number between 18 and 65.
- role:
  - Must be either 'user' or 'admin'.
  - Should default to 'user' if not provided.

```
const userSchema = new mg.Schema({
  username: {
    type: String,
    required: [true, 'Username is required'], // Custom error message
    minlength: [4, 'Username must be at least 4 characters long'],
    maxlength: [20, 'Username cannot be more than 20 characters long'],
    match:[/^[A-Za-z]+[0-9]+$/,"Must starts with letters and ends with digits"],
    trim:true, uppercase:true
  },
  email: {
    type: String,
    unique: [true, 'email already exists'],
    required: [true, 'Email is required'],
    match: [/\S+@\S+\.\S+/, 'Please enter a valid email address']
  },
  age: {
    type: Number,
    min: [18, 'Age must be at least 18'],
    max: [65, 'Age must be less than 65']
  },
  role: {
    type: String,
    enum: ['user', 'admin'], // Value must be either 'user' or 'admin'
    default: 'user'
  }});
```

Task -2: How do you use Mongoose to update or delete documents in a MongoDB collection by name or ID, with options to upsert if the document doesn't exist, and how do you handle errors during these operations?

This question covers the key aspects of the code:

1. Connecting to a MongoDB database using Mongoose.
2. Creating a schema and model.
3. Updating documents by name and ID with upsert options.
4. Deleting a document by ID.
5. Handling errors during these operations.

```
const mg = require("mongoose");
mg.connect("mongodb://127.0.0.1:27017/lju1")
  .then(() => { console.log("success"); })
  .catch((err) => { console.error(err); });
const personSchema = new mg.Schema({  name: String,  age: Number,  active: Boolean});
const Person = mg.model("Person", personSchema);

                                                    //If not want to define scheme for existing record
                                                  //const collection = mg.connection.collection("people");

const updatePerson = async (name, update) => {
  try {
    const result = await Person.updateOne({ name }, update,{upsert:true});
                                        //Use to update record if scheme is not defined for rexisting record
                                              // const result = await collection.updateOne({name:"ABC"},
                                                                        name:"ABC111"},{upsert:true});

    console.log('Update Result:', result);
  } catch (err) { console.error('Error updating person:', err); }
};
const updatePersonById = async (id, update) => {
  try {
    const updatedPerson = await Person.findByIdAndUpdate(id, update, { new: true, upsert: true
});
    console.log('Updated Person:', updatedPerson);
  } catch (err) { console.error('Error updating person:', err); }
};
const deletePersonById = async (id) => {
  try {
    const deletedPerson = await Person.findByIdAndDelete(id);
    if (deletedPerson) {console.log('Deleted Person:', deletedPerson); }
    else {console.log('Person not found');}
  } catch (err) {console.error('Error deleting person by ID:', err);}};
updatePersonById("66c0fd3453eced83156cb23d", { name:"ABC" ,age: 28, active: false });
updatePerson("test", { age: 34, branch:"CE",active: false });
deletePersonById("66c0fd3453eced83156cb23c");
```

Mongoose Example using push method: Insert/Find/Update /Delete

Create Collection “employees” with following data

```
[{_id: 1,name: "Eric",age: 30,position: "Full Stack Developer",salary: 60000},
{_id: 2,name: "Erica",age: 35,position: "Intern",salary: 8000},
{_id: 3,name: "Erical",age: 40,position: "UX/UI Designer",salary: 56000},
{_id: 4,name: "treric7",age: 37,position: "Team Leader",salary: 85000},
{_id: 5,name: "Eliza",age: 25,position: "Software Developer",salary: 45000},
{_id: 6,name: "Trian",age: 29,position: "Data Scientist",salary: 75000},
{_id: 7,name: "Elizan",age: 25,position: "Full Stack Developer",salary: 49000}]
```

1) Find All Documents:
2) Find Documents by Position “Full Stack Developer”:
3) Retrieve name of employees whose age is greater than or equal to 25 and less than or equal to 40.
4) Retrieve name of the employee with the highest salary.
5) Retrieve employees with a salary greater than 50000.
6) Retrieve employees' names and positions, excluding the "_id" field.
7) Count the number of employees who have salary greater than 50000
8) Retrieve employees who are either " **Software Developer**" or "**Full Stack Developer**" and are below 30 years.
9) Increase the salary of an employee who has salary less than 50000 by 10%.
10) Delete all employees who are older than 50.
11) Give a 5% salary raise to all "**Data Scientist**"
12) Find documents where name like “%an”
13) Find documents where name like “Eri--” (Case Insensitive)
14) Find documents where name like “%ric%”
15) Find documents where name contains only 4 or 5 letters.
16) Find documents where name must end with digit

The ***push*() *method*** adds new items to the end of an array. The ***push*() *method*** changes the length of the array. The ***push*() *method*** returns the new length. It will display all results in array.

```
const mongoose = require('mongoose');

// Connect to MongoDB
mg.connect("mongodb://127.0.0.1:27017/lju", { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("Connected to MongoDB"))
  .catch((err) => console.error("Connection error:", err));

// Define the schema
const mySchema = new mg.Schema({
  _id: Number,
  name: { type: String, required: true },
  age: { type: Number, required: true },
  position: { type: String, required: true },
```

```
  salary: { type: Number, required: true }
});

const emp = mg.model("employ", mySchema);

const createDoc = async () => {
  try {
    // Insert multiple documents
    const personData1 = [
      { _id: 1, name: "Eric", age: 30, position: "Full Stack Developer", salary: 60000 },
      { _id: 2, name: "Erica", age: 35, position: "Intern", salary: 8000 },
      { _id: 3, name: "Erical", age: 40, position: "UX/UI Designer", salary: 56000 },
      { _id: 4, name: "treric7", age: 37, position: "Team Leader", salary: 85000 },
      { _id: 5, name: "Eliza", age: 25, position: "Software Developer", salary: 45000 },
      { _id: 6, name: "Trian", age: 29, position: "Data Scientist", salary: 75000 },
      { _id: 7, name: "Elizan", age: 25, position: "Full Stack Developer", salary: 49000 }
    ];

    const result = [];

    // Insert documents
    result.push(await emp.insertMany(personData1));

    // Find all documents
    result.push(await emp.find());

    // Find documents with specific position
    result.push(await emp.find({ position: "Full Stack Developer" }));

    // Find documents within a specific age range
    result.push(await emp.find({ age: { $gte: 30, $lte: 40 } }, { name: 1, _id: 0 }));

    // Find the document with the highest salary
    result.push(await emp.find().sort({ salary: -1 }).limit(1));

    // Find documents with salary greater than 50000
    result.push(await emp.find({ salary: { $gt: 50000 } }));

    // Count documents with salary greater than 50000
    result.push(await emp.find({ salary: { $gt: 50000 } }).countDocuments());

    // Find documents where position is either Software Developer or Full Stack Developer and age is less than 30
    result.push(await emp.find({
      $and: [
        { $or: [{ position: "Software Developer" }, { position: "Full Stack Developer" }] },
        { age: { $lt: 30 } }
      ]    }));
```

```
   // Increase salary by 10% for documents with salary less than 50000
   result.push(await emp.updateMany({ salary: { $lt: 50000 } }, { $mul: { salary: 1.1 } }));

   // Delete documents with age greater than 50
   result.push(await emp.deleteMany({ age: { $gt: 50 } }));

   // Increase salary by 5% for documents with position Data Scientist
   result.push(await emp.updateMany({ position: "Data Scientist" }, { $mul: { salary: 1.05 } }));

   // Find documents where name ends with 'an'
   result.push(await emp.find({ name: { $regex: /an$/i } }));

   // Find documents where name starts with 'eri' and followed by exactly two more characters
   result.push(await emp.find({ name: { $regex: /^eri.{2}$/i } }));

   // Find documents where name contains 'ric' (case insensitive)
   result.push(await emp.find({ name: { $regex: /ric/i } }));

   // Find documents where name has length between 4 and 5 characters (case insensitive)
   result.push(await emp.find({ name: { $regex: /^[A-Za-z]{4,5}$/i } }));

   // Find documents where name ends with a digit
   result.push(await emp.find({ name: { $regex: /[0-9]$/ } }));

   console.log('Query Results:', result);  }
catch (err) {
   console.error("Error occurred:", err);  }
};
createDoc()
```