**Mongoose**

**Elegant MongoDB object modeling for Node.js.**

Mongoose is an ODM (Object Data Modeling) library for MongoDB. While you don’t need to use an Object Data Modeling (ODM) or Object Relational Mapping (ORM) tool to have a great experience with MongoDB. Many Node.js developers choose to work with Mongoose to help with data modeling, schema enforcement, model validation, and general data manipulation. And Mongoose makes these tasks effortless.

**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**

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)});

**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

},

Surname:String,

age:Number,

active:Boolean,

date:{

type:Date,

default:new Date()

}

}

);

With schemas, we define each field and its data type. Permitted types are:

String,Number,Date,Buffer,Boolean,Mixed,ObjectId,Array,Decimal128,Map

**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**

**mg.pluralize(null)** // to add collection as we have mentioned

**Inserting data**

Now that we have our first model and schema set up, we can start inserting data into our database.

Back in the file, let’s insert a new data.

const createDoc=async()=>

{

try{

const personData=new person(

{

name:"def",

Surname:"test",

age:31,

active:true,

email:"abc@gmail.com"

}

)

const result=await personData.save();

}

}

createDoc()

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

**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.

**Example1:**

Write a node.js script to enter 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.

const mg=require("mongoose")

**mg.connect**("mongodb://127.0.0.1:27017/lju").then(()=>{console.log("success")}).catch((err)=>{console.error(err)});

//mg.pluralize(null) The mg.pluralize(null) line is commented out. If uncommented, it would disable Mongoose's automatic pluralization of the collection name. By default, Mongoose pluralizes the model name (e.g., person to people). Uncommenting this line would ensure that the collection name remains person instead of being pluralized to people.

const mySchema=**new mg.Schema**(

{

name:{

type:String,

required:true

},

Surname:String,

age:Number,

active:Boolean,

date:{

type:Date,

default:new Date().toLocaleDateString()

}

}

);

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

const **createDoc**=**async()**=>

{

try{

const **personData**=**new person**({

name:"test",

Surname:"XYZ",

age:3,

active:true

})

const result=**await** **personData**.**save()**; //for single data record

console.log(result);

}

catch(err)

{

console.log("Error Occured" + err);

}

}

**createDoc();**

**Explanation of Example 1**

**Step 1: Importing Mongoose and Connecting to MongoDB**

* **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.
* 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.

### **Step 3: Creating a Model**

* **const person = new mg.model("person", mySchema);:** This creates a Mongoose model named "person" using the defined schema (mySchema).
* The model is used to interact with the person collection in the MongoDB database.

### **Step 4: Creating and Saving a Document**

* **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.

### **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.

**Why to use async and await?**

Here, result is a Promise, and since the save() operation is asynchronous, console.log(result) is called before the Promise is resolved. That's why you see Promise { <pending> } as the output if you do not use async-await.

### **How to Fix It**

To correctly handle the asynchronous operation and see the result of the save() method, you should use await or handle the Promise using .then().

**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.**

**Insert by creating an Instance**

const mg=require("mongoose")

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

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 personData=new person(**

**{**

**name:"test",**

**Surname:"test1",**

**age:33,**

**active:true**

**}**

**)**

**const personData1=new person(**

**{**

**name:"hi",**

**Surname:"hi1",**

**age:30,**

**active:true**

**}**

**)**

**const personData2=new person(**

**{**

**name:"hello",**

**Surname:"hello1",**

**age:37,**

**active:true**

**}**

**)**

**const personData3=new person(**

**{**

**name:"hello",**

**Surname:"hello11",**

**age:37,**

**active:true**

**}**

**)**

**const result= await person.insertMany ([personData,personData1,personData2, personData3])**

console.log(result)

}

catch(err)

{

console.log("problem");

}

}

**createDoc();**

**OR**

**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)});

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)

}

catch(err){console.log("problem");}

}

createDoc();

**updateOne, findByIdAndUpdate, and findByIdAndDelete**

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:**

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);

const **updatePerson** = **async (name, update)** => {

  try {

    const result = await Person.**updateOne**({ name }, update,{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:"LJU" ,age: 28, active: false });

**updatePerson**("test", { age: 34, branch:"CE",active: false });

**deletePersonById**("66c0fd3453eced83156cb23c");

**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.

### **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.

### **Example**

**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.**

1. **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 mg = require('mongoose');

**mg.connect**("mongodb://127.0.0.1:27017/validations").then(()=>{console.log("success")}).catch((err)=>{console.error(err)});

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 start with letters and end 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'

  }

});

const User = **mg.model**('User', userSchema);

const **createDoc**=async()=>

    {

        try{

            const **newUser** = new User({

                username: '  sfga34 ',

                email: ' a1e@xample.com',

                age: 25,

                role: 'user'

              });

            const result= await **newUser**.save(); **//for single data record**

            console.log(result);

        }

        catch(err)

        {

            console.log("Error Occured" + err);

        }

    }

**createDoc();**

**validator**

The validator package is a popular library in Node.js that provides a variety of string validation and sanitization functions. When used with Mongoose, it allows you to apply these validations to your schema fields to ensure that the data being stored in MongoDB is valid and meets specific criteria.

**Installation of validator module:**

**npm install validator**

**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(),isAlphanumeric() etc.

**Example:**

**You are developing a MongoDB application using Mongoose and need to enforce specific validation rules on the email and product fields in your userSchema.**

1. **Define a Mongoose schema userSchema with the following fields:**
   * **email:**
     + **The field is required and must be unique in the database.**
     + **It should be validated to ensure it contains a valid email address format.**
     + **If the provided email is invalid, the error message should indicate that the email address is not valid.**
   * **product:**
     + **The field is required.**
     + **It should only allow alphanumeric characters (i.e., letters and numbers).**
     + **If the product field contains invalid characters, the error message should indicate that it is not a valid alphanumeric code.**
2. **Write an asynchronous function createDoc that creates and saves a document with random data.**

**const mg = require('mongoose');**

**const validator = require('validator');**

**mg.connect**("mongodb://127.0.0.1:27017/validations").then(()=>{console.log("success")}).catch((err)=>{console.error(err)});

const userSchema = new **mg.Schema**({

  email: {

    type: String,

    required: true,

    unique: true,

**validate**: { **validator**: (v) => {return **validator**.**isEmail**(v);},

**message**: `This is not a valid email address!`

    }

  },

  product: {

    type: String,

    required: true,

**validate**: {

**validator**: (v) => {return **validator**.**isAlphanumeric**(v);},

**message**: `This is not a valid alphanumeric code!`

    }

  }

});

const **User** = **mg.model**('User', userSchema);

const createDoc=async()=>

    {

        try{

            const **newUser** = new **User**({

                email: 'age@gmail.com',

                product:"fhxvf111"

              });

            const result= await **newUser**.**save()**; //for single data record

            console.log(result);

        }

        catch(err)

        {

            console.log("Error Occured" + err);

        }

    }

    createDoc();

**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 mongoose = require('mongoose');**

**const validator = require('validator');**

**// Connect to MongoDB**

**mongoose.connect**("mongodb://127.0.0.1:27017/validations")

  .then(() => console.log("Connected to MongoDB"))

  .catch((err) => console.error("Connection error:", err));

**// Define the schema**

const **userSchema** = new **mongoose.Schema**({

  name: {

    type: String,

    required: [true, 'Name is required'],

    minlength: [3, 'Name must be at least 3 characters long'],

    maxlength: [10, 'Name cannot be more than 10 characters long'],

    trim: true,

lowercase:true

  },

  age: {

    type: Number,

    required: [true, 'Age is required'],

    min: [1, 'Age must be greater than 0']

  },

  email: {

    type: String,

    required: [true, 'Email is required'],

    unique: true,

    validate: {

      validator: function(v) { return **validator**.isEmail(v); },

      message: props => `${props.value} is not a valid email address!`

    }

  },

  gender: {

    type: String,

    required: [true, 'Gender is required'],

    uppercase: true, **// Converts the gender to uppercase before saving**

    enum: {

      values: ['MALE', 'FEMALE'],

      message: '{VALUE} is not a valid gender! Allowed values are MALE or FEMALE'

    }

  }

});

**// Create the model**

const User = **mongoose.model**('User', **userSchema**);

**// Function to create a document**

const createDoc = async () => {

  try {

    const **newUser** = new User({

      name: '  LjUuser  ', **// Leading and trailing spaces will be trimmed, and name will be stored in lowercase**

      age: 25,

      email: 'lju@example.com',

      gender: 'male' **// This will be converted to 'MALE'**

    });

    const result = await **newUser**.**save();** **// Save the document**

    console.log(result);

  } catch (err) {

    console.error("Error occurred:", err.message);

  }

};

// Call the function to create a document

createDoc();

**Mongoose Example using push method**

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. Count the number of employees who have salary greater than 50000
7. Retrieve employees who are either " **Software Developer**" or "**Full Stack Developer**" and are below 30 years.
8. Increase the salary of an employee who has salary less than 50000 by 10%.
9. Delete all employees who are older than 50.
10. Give a 5% salary raise to all "**Data Scientist**"
11. Find documents where name like “%an”
12. Find documents where name like “Eri--” (Case Insensitive)
13. Find documents where name like “%ric%”
14. Find documents where name contains only 4 or 5 letters.
15. 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**();

**Connectivity of MongoDB with ExpressJS/NodeJS using HTML**

Install express if not already installed using below command.

**npm install express**

**Example:**

**Create a form having username and password and insert data entered by user in collection named “data1”.**

**task.js**

**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} })

const **person** =new **mg.model**("data1", myschema)

**app.use(expr.static(\_\_dirname,{index:"form.html"}))**

app.get("/process\_get",async (req,res)=>{

const **personData**=new **person**({

uname:req.query.uname,

password:req.query.pwd

})

**await personData.save()**

res.send("Record inserted")

})

app.listen(6000)

**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>

**To run**

**In terminal node task.js**

**In browser localhost:6000**

### **How It Works:**

1. When the server is running and a user navigates to the root URL (e.g., http://localhost:6000), the form.html page is served.
2. The user fills in the username and password fields in the form and submits it.
3. The form data is sent to the server via a GET request to the /process\_get endpoint.
4. The server extracts the data from the query parameters, creates a new document using the person model, and saves it to the data1 collection in the login database.
5. Upon successful insertion, the server responds with the message "Record inserted".

### **Running the Application:**

* **Start the server:** Run node task.js in the terminal. Ensure that MongoDB is running on your machine.
* **Access the form:** Open a web browser and navigate to http://localhost:6000. You will see the form where you can enter a username and password.
* **Submit the form:** After submitting, if everything is set up correctly, you'll see the message "Record inserted," and the data will be saved to the MongoDB database.

**React-express/node-mongodb**

**Frontend**

**Create a React Client**

**npx create-react-app myapp**

cd myapp

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

**npm install axios**

**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

});

**myapp/src/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('Welcome ' + 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}>**

**<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;**

**To run**

npm start

**Backend**

**Dependencies:**

* **express**: A web framework for Node.js used to create the server. **(npm i express)**
* **mongoose**: A MongoDB object modeling tool designed to work in an asynchronous environment. **(npm i mongoose)**
* **cors**: A middleware to allow cross-origin requests. **(npm icors)**

**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.

**express/server.js**

**const express = require('express');**

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

**const cors = require('cors');**

**const app = express();**

**app.use(cors());**

**app.use(express.json());**

**mongoose.connect('mongodb://127.0.0.1:27017/reactconnect');**

**const UserSchema = new mongoose.Schema({**

**username:String**

**});**

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

**app.post('/signup', async (req, res) => {**

**try {**

**const { username} = req.body;**

**const newUser = new User({ username});**

**await newUser.save();**

**res.send();**

**} catch (error) {**

**res.send(error);**

**}**

**});app.listen(5000);**

**To run**

express> node server.js

### **How It Works:**

1. A user fills in their username on the frontend and submits the form.
2. The React app sends a POST request to the Express server with the username.
3. The Express server receives the request, creates a new user document in the MongoDB database, and saves the username.
4. If successful, the React app displays a welcome message with the username and clears the input field.

### **Running the Application:**

* **Frontend:**
  + Start the React app by running npm start in the terminal. This will start the development server for the React app.
* **Backend:**
  + Start the Express server by running node server.js in the terminal. This will connect to MongoDB and start listening on port 5000 for incoming requests.

The frontend and backend need to be running simultaneously, with the React app on the default port (3000) and the Express server on port 5000.