# 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
```

**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.

**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.

## 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/zarana1").then(()=>{console.log("succe
ss")}).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().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();
```

## 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)});
//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 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)});
//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)
      }
         catch(err)
         {
            console.log("problem");
         }
   }
createDoc();
```

## Update /delete by ID

- **updateOne()**: Is meant to perform an atomic update operation against "one or more" documents matched by it's query condition in a collection. It returns the number of modified documents in it's response.
- **findOneAndUpdate()**: Has the purpose of both processing an update statment on a "singular" document, as well as retrieving the content of that "singular" document. The state returned depends on the value of the "**new**" option as passed to the operation. Where true the "modified" document is returned. Where false the "original" document is returned before any modification. The latter form is the default option.

## Example:

**Write a node.js script to update a specific document using an _id field. Display the updated result on console. (using updateOne and findByIdAndUpdate method)**

**Write a node.js script to delete a specific document using an _id field.**

```
const mg=require("mongoose")
mg.connect("mongodb://127.0.0.1:27017/zarana1").then(()=>{console.log("succe
ss")}).catch((err)=>{console.error(err)});
//mg.pluralize(null)
const mySchema=new mg.Schema(
   {
       name:String,
       Surname:String,
       age:Number    }
);
const person=new mg.model("person",mySchema)
const updateDoc=async(i) =>
 {
         const result=await person.updateOne({_id:i},
         {
              $set:{age:37}
         })
```

```
        const result=await person.findByIdAndUpdate({_id:i},
        {
            $set:{age:37}
        },{new:true})

        const result=await person.findByIdAndDelete({_id:i})
        console.log(result)
 }
updateDoc("64a7cd53376fc04b3c2637bd");
```

## Mongoose Schema validation

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.

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.

**Installation of validator module:**

```
npm install validator
```

## 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:[10,"max length must be 10"]
      },
      age:{
         type:Number,
         validate(v1){
            if(v1<=0)
            {
               let k=new Error("Age must be positive");
               throw k;
            }
         }
      },
      email: {
         type:String,
         validate(val)
         {
            if(!v.isEmail(val))
            {
               throw new Error("Enter valid email_id")
            }
         }
      },
      gender: {
         uppercase: true,
         enum:['MALE','FEMALE']
      }
   }
);
const person=new mg.model("person",mySchema)
const createDoc=async()=>
   {
    try {
      const personData=new person(
         {
            name:"def",
            age:31,
                email:def@gmail.com,
                gender:"FEMALE"
            }
        )
        const personData1=new person(
            {
                name:"Abc",
                age:26,
                email:abc@gmail.com,
                gender:"MALE"
            }
        )

        const result= await person.insertMany([personData,personData1])
        console.log(result)
       }
       catch(err)
        {
                console.log("problem");
         }
   }
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) 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 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( {
         _id:Number,    name:String,
         age:Number,    position:String,
         salary:Number
```

```
    });
const emp=new mg.model("employ",mySchema)
const createDoc=async()=>
  {
    try{
      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=[]
     result.push(await emp.insertMany(personData1))
     result.push(await emp.find())
     result.push(await emp.find({position:"Full Stack Developer"}))
     result.push(await emp.find({ age: { $gte: 30, $lte: 40 } },{name:1,_id:0}))
     result.push(await emp.find().sort({ salary: -1 }).limit(1))
     result.push(await emp.find({ salary: { $gt: 50000 } }))
     result.push(await emp.find({salary:{$gt:50000}}).count())
     result.push(await emp.find({ $and: [{ $or: [{ position: "Software Developer" }, { position: "Full Stack Developer" }] }, { age: { $lt: 30 } }] }))
     result.push(await emp.updateOne({salary:{$lt:50000}},{ $mul: { salary: 1.1 } }))
     result.push(await emp.deleteMany({ age: { $gt: 50 } }))
     result.push(await emp.updateMany({ position: "Data Scientist" }, { $mul: { salary: 1.05 } }))
     result.push(await emp.find({name:{$regex:/an$/}}))
     result.push(await emp.find({name:{$regex:/^eri[A-z]{2}$/i}}))
     result.push(await emp.find({name:{$regex:/ric/i}})
     result.push(await emp.find({name:{$regex:/^[A-Za-z]{4,5}$/i}})
     result.push(await emp.find({name:{$regex:/[0-9]$/}})
      console.log('Query Results:', result);
            }
     catch(err)     {
        console.log(err);
     } }
createDoc()
```

## Connectivity of MongoDB with ExpressJS/NodeJS

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"}))
```

```
//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(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**

# Connect MongoDB with ReactJS

**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**.

**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.');
      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>
    </form>
   </div>
  );
}

export default Signup;
```

**App.js**

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

**To run 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.

**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
npm install body-parser
```

- Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node. It provides a simplified schema-based method, to model your application data and store it in a MongoDB database.

- The CORS (Cross-Origin Resource Sharing) package provides a mechanism for the backend server and a frontend client to communicate and pass data via the API endpoints.

- **app.use(bodyParser.json())** basically tells the system that you want json to be used. Here we are using .json

- **bodyParser.urlencoded({extended: ...})** basically tells the system whether you want to use a simple algorithm for shallow parsing (i.e. false) or complex algorithm for deep parsing that can deal with nested objects (i.e. true).

**server.js**

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const cors = require('cors');
const app = express();

app.use(cors());
app.use(bodyParser.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);
```

**Tu run**

**node server.js**

## Example:

**Write a script using frontend technology to have a email field and a submit button. After clicking submit button, insert that value of email in database.**

**exprs.js**

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const cors = require('cors');
const app = express();

app.use(cors());
app.use(bodyParser.json());

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

```
const UserSchema = new mongoose.Schema({
  eid:String
});

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

app.post('/process', async (req, res) => {
  try {
    const { eid} = req.body;
    const newUser = new User({ eid });
    await newUser.save();
    res.send();
  } catch (error) {
    res.send(error);
  }
});

app.listen(8000);
```

**Tu run**

**node exprs.js**

**Form.js**

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

function Form() {
  const [eid, setEmailid] = useState('');

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

    try {
      await axios.post('http://localhost:8000/process', {
        eid
      });
     alert('User signed up successfully.');
      setEmailid('');

    } catch (error) {
      console.error('Error signing up:', error);
     alert('An error occurred.');
    }
  };

  return (
    <div>
      <form onSubmit={handleSignup}>
        <input
          type="email"
          placeholder="Email id"
          value={eid}
          onChange={(e) => setEmailid(e.target.value)}
        />
        <button type="submit">Sign Up</button>
      </form>
    </div>
  );
}

export default Form;
```