# Learning Outcome

After completing this module, the student should be able to configure an embedded Mongo DB and Create application with CRUD operation

To meet the learning outcome, a student has to complete the following activities

1. Create application with CRUD operations using Mongo DB (2Hr)

# Activity 1

## Aim: Create application with CRUD operations using Mongo DB

**Learning outcome:** Able to configure embedded Mongo DB application with Node JS

**Duration:** 2 hour

**List of Hardware/Software requirements:**

1. Laptop/Computer with Windows 10/11
2. Mongo DB
3. Node JS, Curl

**Node JS and MongoDB CRUD Operations**

In this activity, we will be discussing how to run MongoDB CRUD operations from Node.js.

**Method 1**

### **1. Node JS with MongoDB CRUD Operations: MongoDB and Node.js Setup**

Your computer should be installed with MongoDB and Node.js and a command-line with curl command. Download MongoDB MSI package from its official website and run it to install MongoDB. After the installation, run the mongo command to ensure that MongoDB is running.

Next, ensure that both MongoDB and npm are installed on your computer. You can download Node.js from its official website and install it on your computer. Curl will help you to run HTTP requests on the command line.

### **2. Node JS with MongoDB CRUD Operations: Create a New Node.js Project**

Open the folder where you will create a project and type the following command:

npm init

Give the project the name node-mongo and accept defaults for the other settings. Add the necessary dependencies to the project. Run the following command within the project directory:

npm install mongodb polka --save

The above command will install the Node.js driver for MongoDB and the Polka HTTP Server to handle HTTP requests. Add the following start script to the package.json file:

Create a**/node-mongo/src/index.js** file and add the following contents to it:

const polk= require('polka');

polk()

.get('/create', (req, res) => {

res.end(`It works`);

})

.listen(3000, err => {

if (err) throw err;

console.log(`> localhost:3000`);

});

Now, run the following command to start the server:

npm run start

Test the server by running the following command:

curl http://localhost:3000/create

### **3. Node JS with MongoDB CRUD Operations: Inserting a Record into GridDB**

We need to perform an insert, which is the C in CRUD. Change your index.js file to the following:

const polk = require('polka');

const { MongoClient } = require("mongodb");

polk()

.get('/create', (req, res) => {

const cl = new MongoClient("mongodb://localhost:27017");

async function run() {

try {

await cl.connect();

const dbs = client.db("intro");

const coll = dbs.collection("quotes");

const rest = await coll.insertOne({"quote":"This is my quote."});

res.end(JSON.stringify(rest));

} catch (ex) {

console.log("Error: " + ex);

} finally {

await cl.close();

}

}

run().catch(console.dir);

})

.listen(3000, err => {

if (err) throw err;

console.log(`> localhost:3000`);

});

The above script is an example of a node js with MongoDB crud operation. It establishes a connection to a MongoDB instance, selects a database named intro and a collection named quotes. It then inserts a document into the collection and returns the results as an HTTP response.

To run the insert, press Ctrl+C to stop and restart the node. Next, run the following command on the terminal:

npm run startcurl http://localhost:3000/create

### **4. Node JS with MongoDB CRUD Operations: Retrieving from MongoDB**

We can use the .get() method to retrieve a document from MongoDB. The following script demonstrates this:

.get('/retrieve', (req, res) => {

const cl = new MongoClient("mongodb://localhost:27017");

async function run() {

try {

await cl.connect();

const dbs= client.db("intro");

const coll = dbs.collection("quotes");

const cur = coll.find({}, {});

let items = [];

await cur.forEach(function(doc){

items.push(doc);

});

res.end(JSON.stringify(items));

} catch (err){

console.warn("ERROR: " + err);

if (errCallback) errCallback(err);

} finally {

await cl.close();

}

}

run().catch(console.dir);

})

We have used the find command and passed to it an empty query. This means that it will match all documents in the collection. The response is then returned to the client in the form of an array.

### **5. Node JS with MongoDB CRUD Operations: Updating MongoDB Documents**

In this section, we will demonstrate how to **update** MongoDB documents, which is the “U” in CRUD:

.get('/update', (req, res) => {

const cl = new MongoClient("mongodb://localhost:27017");

async function run() {

try {

await cl.connect();

const dbs = client.db("intro");

const coll = dbs.collection("quotes");

const updateDocument = {

$set: {

author:

"Martin Kings",

},

};

const rst = await coll.updateOne({}, updateDocument, {});

res.end("Updated: " + rst.modifiedCount);

} catch (ex) {

errCallback(ex);

} finally {

await cl.close();

}

}

run().catch(console.dir);

})

In the above node js with MongoDB crud operation, we have connected to the database and created an updated document. We have then changed the author field to Martin Kings. The updateOne() function helped us to execute the update. Since we wanted to match all documents, the filter is empty.

### **6. Node JS and MongoDB CRUD Operations: Deleting MongoDB Documents**

The following script demonstrates how to delete a MongoDB document from Node.js:

.get('/delete', (req, res) => {

const cl = new MongoClient("mongodb://localhost:27017");

async function run() {

try {

await cl.connect();

const dbs = cl.db("intro");

const coll = dbs.collection("quotes");

const qry = { };

const rst = await coll.deleteOne(qry);

if (rst.deletedCount === 1) {

res.end("One document deleted.");

} else {

res.end("No document was deleted.");

}

} finally {

await cl.close();

}

}

We have used an empty query so as to match all the documents within the collection. The deleteOne() function tells us the number of documents that have been deleted.

That is how to run node js with MongoDB crud operations.

**Reference:**

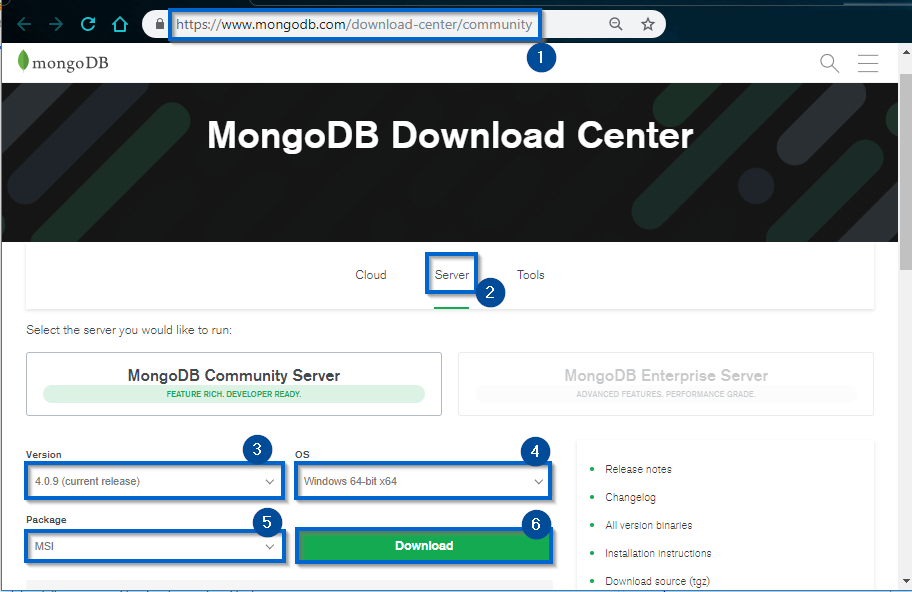
<https://hevodata.com/learn/node-js-with-mongodb-crud/>

**Method 2**

### **MongoDB Installation**

**Step I:**Download the latest version MongoDB server from its official site: <https://www.mongodb.com/download-center/community>

**Step II:**Next click on the ‘Server’ tab as shown in the below screenshot.



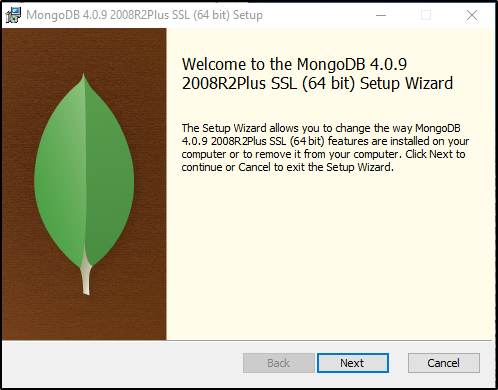
**Step III:**If you are looking for any specific version, you can select it from the drop-down list or you can just download the latest version.

**Step IV:** Select your OS from the drop down. Since I am working on Windows I will go for Windows 64 bit.

**Step V:**Now, select the package as MSI.

**Step VI:**Finally, click on ‘Download’ to begin the download process.

**Step VII:**Once downloaded, double click on the MSI file to open it and proceed with the installation wizard.

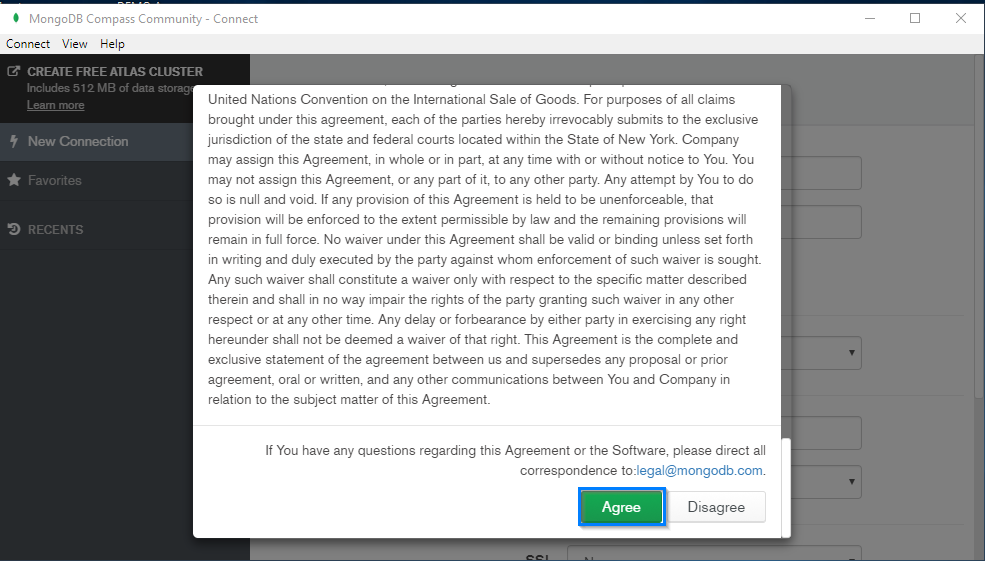


**Step VIII:**Now, in order to start the MongoDB server you have to run the .exe file and assign the database folder. To make the work easier, all you need to do is write down a few lines of code in a notepad file and save it with the .bat extension. In other words, you just need to create a batch file, which will start the MongoDB server for you without any hassle. To create the batch file type in the below code:

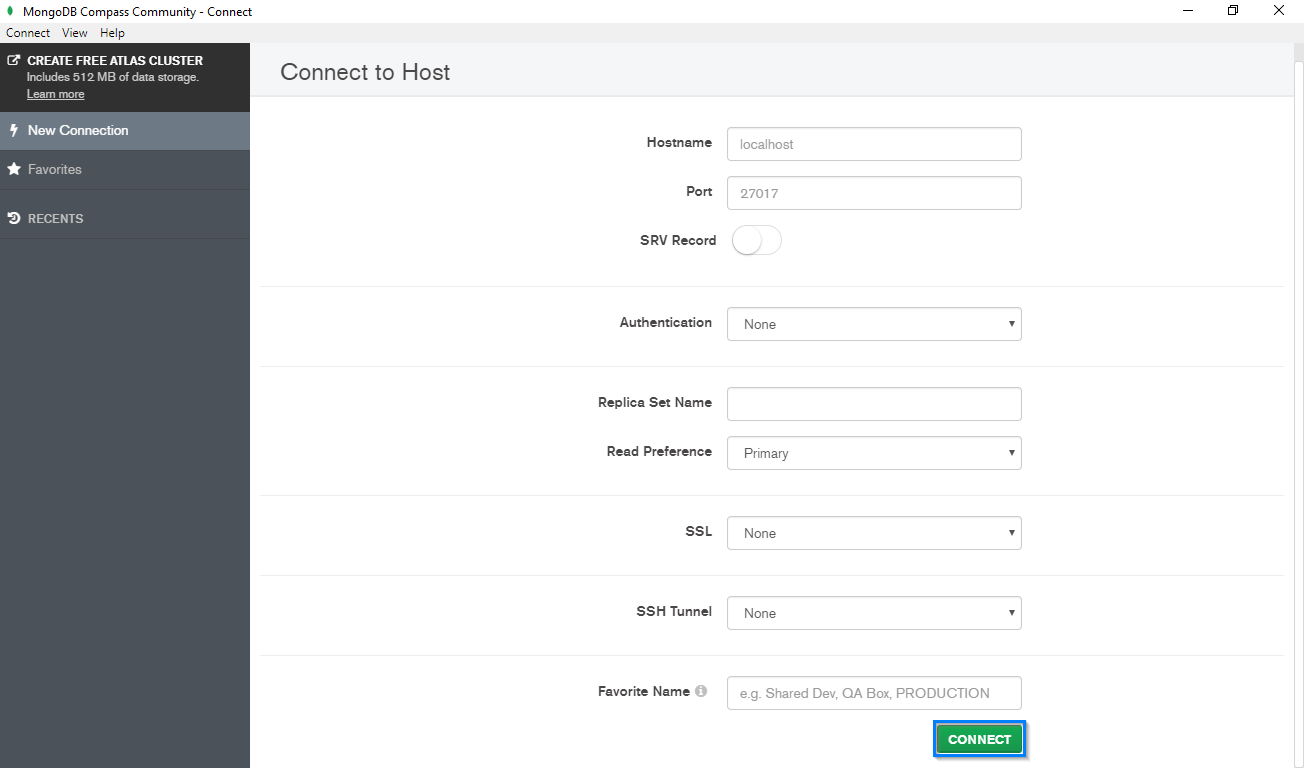
|  |  |
| --- | --- |
| 1  2 | cd C:Program FilesMongoDBServer.0in (MongoDB path)  mongod.exe --dbpath F:MongoDBdata (database dump destination) |

Now, whenever you want to launch the MongoDB server, all you need to do is double click this batch file and open the MongoDB Compass application.

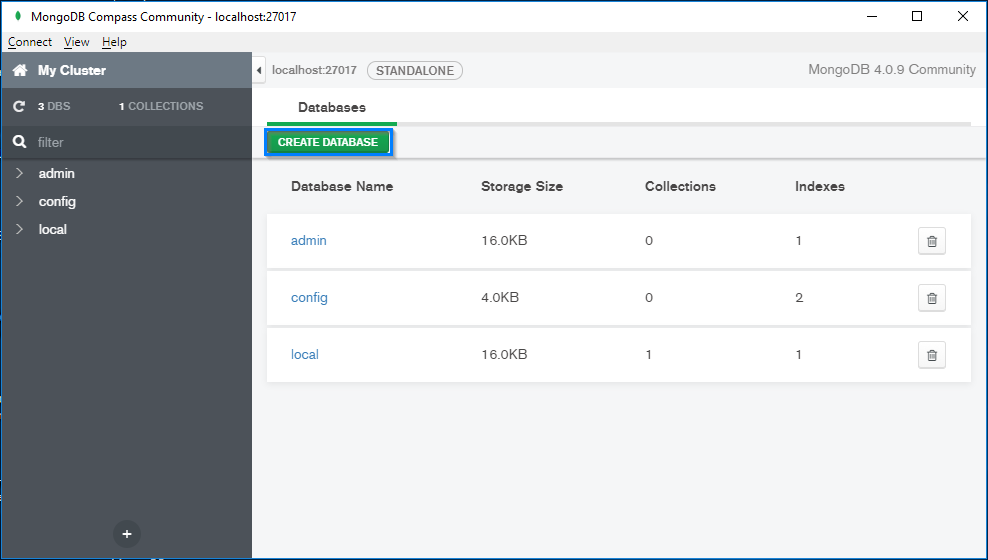
**Step IX:**Next, you need to launch the ‘MongoDB Compass’ and agree to its terms of use.



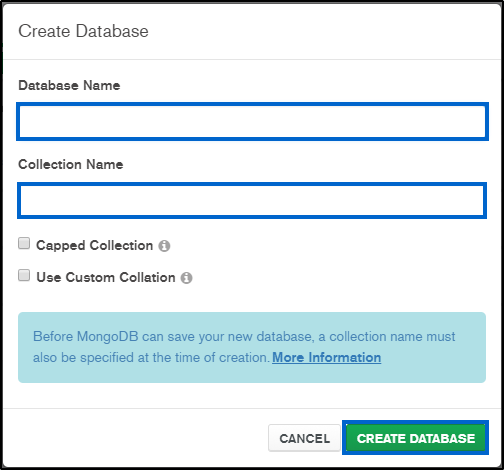
**Step X:**Now you need to provide the server configurations and hit ‘Connect’.



**Step XI:**Next, click on ‘Create Database’.



**Step XII:**Now, provide a relevant name for your database and collection and hit ‘Create Database’.



I guess, now you are all set to get started with the practical part, so without any more delay let’s dive into the code.

## ****Node.js MongoDB Demo****

Here I will be creating a CRUD application for Course Management with the help of [Node.js and Express.js](https://www.edureka.co/blog/rest-api-with-node-js/) and use MongoDB to store the data. In this application, I will be taking course details like name, id, duration, and fee as inputs. For that, I will be creating a few view files which will act as an interface. Then in order to handle the data, I will be needing a controller as well which will help in manipulating the data. Finally, I will be needing a few model files to store the data. So basically, I will be following an MVC pattern for this application development. So, lets now jump into development.

Our application will be having the following hierarchy:

* **NodejsMongoDbDemo**
  + package.json
  + script.js
  + **controllers**
    - courseController.js
  + **img**
    - logo.jpg
  + **models**
    - course.model.js
    - mongodb.js
  + **views**
    - **course**
      * courseAddEdit.hbs
      * list.hbs
    - **layouts**
      * mainLayout.hbs

So, let’s begin the application development by creating a directory for the project. Once you are done, open the command prompt and navigate to your project directory. Now you need to set up the project configurations for that, type in the below command and provide the necessary details:

|  |  |
| --- | --- |
| 1 | npm init |

Now, you need to install the required packages. So, in this project, I am using the below packages:

* **express.js:**It is a web framework.
* **express-handlebars:**It is a template engine and helps in creating client-side applications.
* **mongoose:**Helps in communicating with MongoDB.
* **body-parser:**Helps in converting the POST data into the request body.
* **nodemon:**Helps in automatically restarting the server whenever the code changes.

In order to install these packages, type in the following command:

|  |  |
| --- | --- |
| 1 | npm i --s express express-handlebars mongoose body-parser |

Since I want to install nodemon such that it can access any file in the directory, I will be installing it with the global command:

|  |  |
| --- | --- |
| 1 | npm i -g nodemon |

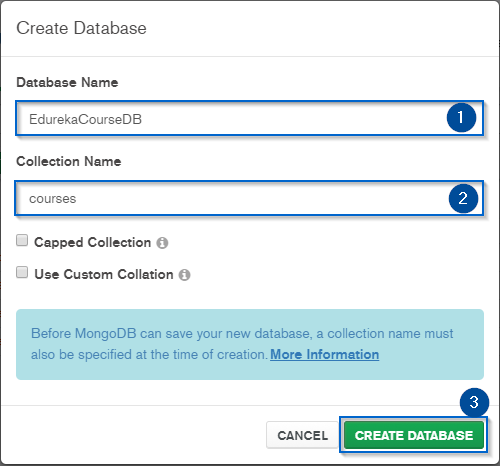
Once you are done installing with the packages, your final JSON file should look like the below file:

**package.json**

EdunetCourseDB

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | {  "name": "samplenodemongo",  "version": "1.0.0",  "description": "Edunet demo on how to build a Node.js application with MongoDB",  "main": "script.js",  "scripts": {  "test": "echo "Error: no test specified" && exit 1"  },  "author": "Edunet",  "license": "ISC",  "dependencies": {  "body-parser": "^1.19.0",  "express": "^4.16.4",  "express-handlebars": "^3.0.2",  "mongoose": "^5.5.6",  "nodemon": "^1.19.0"  }  } |

As you can see, in the dependencies section all the installed packages have been successfully listed. So, lets now create the database we will be using in this demo. For that start the batch file and open MongoDB application. Now, create a new database and provide a collection name. In my application, I will be using ‘EdunetCoursesDB’ as the database name and ‘courses’ as the collection.



Now, switch back to your code editor where we will be creating the files to establish connectivity between Node.js and MongoDB. For that, first, you need to create a folder inside the project directory and name it ‘model’. Inside this folder, create a javascript file with the name ‘**mongodb.js**‘ and type in the below code:

**mongodb.js**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | const mongoose = require('mongoose');  mongoose.connect('mongodb://localhost:27017/EdunetCoursesDB', {useNewUrlParser: true}, (err) => {  if (!err) {  console.log('Successfully Established Connection with MongoDB')  }  else {  console.log('Failed to Establish Connection with MongoDB with Error: '+ err)  }  });    //Connecting Node and MongoDB  require('./course.model'); |

Now, you need to define the schema of your course database. For that, create a new JS file within the model folder and name it ‘**course.model.js**‘. So, I am using four fields in my course object I am using four fields which are name, id, duration, and fee. To create this file, type in the below-given code.

**course.model.js**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | const mongoose = require('mongoose');    //Attributes of the Course object  var courseSchema = new mongoose.Schema({  courseName: {  type: String,  required: 'This field is required!'  },  courseId: {  type: String  },  courseDuration: {  type: String  },  courseFee: {  type: String  }  });    mongoose.model('Course', courseSchema); |

Now, you need to create the root file called ‘**script.js**‘. This file is the entry point of this application and will contain all the connection paths in it. You need to be really careful while providing the paths in this file as it might result in an error or application failure. Along with this, it is also responsible for invoking the server and establish the connection. In order to create this file, type in the below code:

**script.js**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | require('./models/mongodb');    //Import the necessary packages  const express = require('express');  var app = express();  const path = require('path');  const exphb = require('express-handlebars');  const bodyparser = require('body-parser');    const courseController = require('./controllers/courseController');    app.use(bodyparser.urlencoded({  extended: true  }));    //Create a welcome message and direct them to the main page  app.get('/', (req, res) => {  res.send('    <h2 style="font-family: Malgun Gothic; color: midnightblue ">Welcome to Edunet Node.js MongoDB Tutorial!!</h2>     Click Here to go to <b> <a href="/course">Course Page</a> </b>');  });  app.use(bodyparser.json());    //Configuring Express middleware for the handlebars  app.set('views', path.join(\_\_dirname, '/views/'));  app.engine('hbs', exphb({ extname: 'hbs', defaultLayout: 'mainLayout', layoutDir: \_\_dirname + 'views/layouts/' }));  app.set('view engine', 'hbs');    //Establish the server connection  //PORT ENVIRONMENT VARIABLE  const port = process.env.PORT || 8080;  app.listen(port, () => console.log(`Listening on port ${port}..`));   //Set the Controller path which will be responding the user actions  app.use('/course', courseController); |

Next, in order to handle the user requests, you need to create the router file. For that first, create a folder and name it ‘controller’ and within this folder create a file with the name ‘**courseController.js**‘. In this file, we will be dealing with the CRUD operations related to the employee. Below is the code for creating this file:

**courseController.js**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107 | //Import the dependencies  const express = require('express');  const mongoose = require('mongoose');  //Creating a Router  var router = express.Router();  //Link  const Course = mongoose.model('Course');    //Router Controller for READ request  router.get('/',(req, res) => {  res.render("course/courseAddEdit", {  viewTitle: "Insert a New Course for Edunet"  });  });    //Router Controller for UPDATE request  router.post('/', (req,res) => {  if (req.body.\_id == '')  insertIntoMongoDB(req, res);  else  updateIntoMongoDB(req, res);  });    //Creating function to insert data into MongoDB  function insertIntoMongoDB(req,res) {  var course = new Course();  course.courseName = req.body.courseName;  course.courseId = req.body.courseId;  course.courseDuration = req.body.courseDuration;  course.courseFee = req.body.courseFee;  course.save((err, doc) => {  if (!err)  res.redirect('course/list');  else  console.log('Error during record insertion : ' + err);  });  }    //Creating a function to update data in MongoDB  function updateIntoMongoDB(req, res) {  Course.findOneAndUpdate({ \_id: req.body.\_id }, req.body, { new: true }, (err, doc) => {  if (!err) { res.redirect('course/list'); }  else {  if (err.name == 'ValidationError') {  handleValidationError(err, req.body);  res.render("course/courseAddEdit", {  //Retaining value to be displayed in the child view  viewTitle: 'Update Course Details',  employee: req.body  });  }  else  console.log('Error during updating the record: ' + err);  }  });  }    //Router to retrieve the complete list of available courses  router.get('/list', (req,res) => {  Course.find((err, docs) => {  if(!err){  res.render("course/list", {  list: docs  });  }  else {  console.log('Failed to retrieve the Course List: '+ err);  }  });  });    //Creating a function to implement input validations  function handleValidationError(err, body) {  for (field in err.errors) {  switch (err.errors[field].path) {  case 'courseName':  body['courseNameError'] = err.errors[field].message;  break;  default:  break;  }  }  }    //Router to update a course using it's ID  router.get('/:id', (req, res) => {  Course.findById(req.params.id, (err, doc) => {  if (!err) {  res.render("course/courseAddEdit", {  viewTitle: "Update Course Details",  course: doc  });  }  });  });    //Router Controller for DELETE request  router.get('/delete/:id', (req, res) => {  Course.findByIdAndRemove(req.params.id, (err, doc) => {  if (!err) {  res.redirect('/course/list');  }  else { console.log('Failed to Delete Course Details: ' + err); }  });  });    module.exports = router; |

Now, that we are done with backend files, the next step is to create the Views. For that first, you need to create a wrapper for the child views. But before that, create a folder with name ‘**views**‘. Inside this folder create two more folders with names ‘**course**‘ and ‘**layouts**‘ respectively with .hbs extension. Now, navigate inside the ‘layouts’ folder and create the wrapper with the name ‘**mainLayout.hbs**‘. This file will contain the basic skeleton of the application which will be reflected in the child views as well. In this file, I am inserting an image as well, so for that I will create a local folder with name **img**and save my image inside.

To create this file, type in the below codes:

**mainLayout.hbs**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | <!DOCTYPE html>  <html>  <head>  <title>Edunet Node.js MongoDB Demo</title>  <link rel="stylesheet" href="<https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css>" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">  <img src="data:image/gif;base64,R0lGODlhAQABAIAAAAAAAP///yH5BAEAAAAALAAAAAABAAEAAAIBRAA7" data-wp-preserve="%3Cscript%20src%3D%22https%3A%2F%2Fajax.googleapis.com%2Fajax%2Flibs%2Fjquery%2F3.2.1%2Fjquery.min.js%22%3E%3C%2Fscript%3E" data-mce-resize="false" data-mce-placeholder="1" class="mce-object" width="20" height="20" alt="&lt;script&gt;" title="&lt;script&gt;" />  <img src="data:image/gif;base64,R0lGODlhAQABAIAAAAAAAP///yH5BAEAAAAALAAAAAABAAEAAAIBRAA7" data-wp-preserve="%3Cscript%20src%3D%22https%3A%2F%2Fmaxcdn.bootstrapcdn.com%2Fbootstrap%2F3.3.7%2Fjs%2Fbootstrap.min.js%22%3E%3C%2Fscript%3E" data-mce-resize="false" data-mce-placeholder="1" class="mce-object" width="20" height="20" alt="&lt;script&gt;" title="&lt;script&gt;" />  </head>    <body class = "bg-info">      <div align="center">  <!-- Inserting the image -->  <img src="/static/EdunetLogo.png" alt="Edunet Logo">  </div>  <div class="row">    <div class="col-md-6 offset-md-3" style="background-color: #fff; margin-top: 40px; padding:20px;">  <!-- retrieving HTML String from the child Views -->  {{{body}}}  </div>      </div>      </body>  </html> |

Finally, inside the course folder, we will be creating two child views, one of which will be used for course addition or update and the second view will display the complete list of the available courses. Let’s first focus on the first view i.e **courseAddEdit.hbs** which will look like the below screenshot.



**EduNet!**

**Edunet**

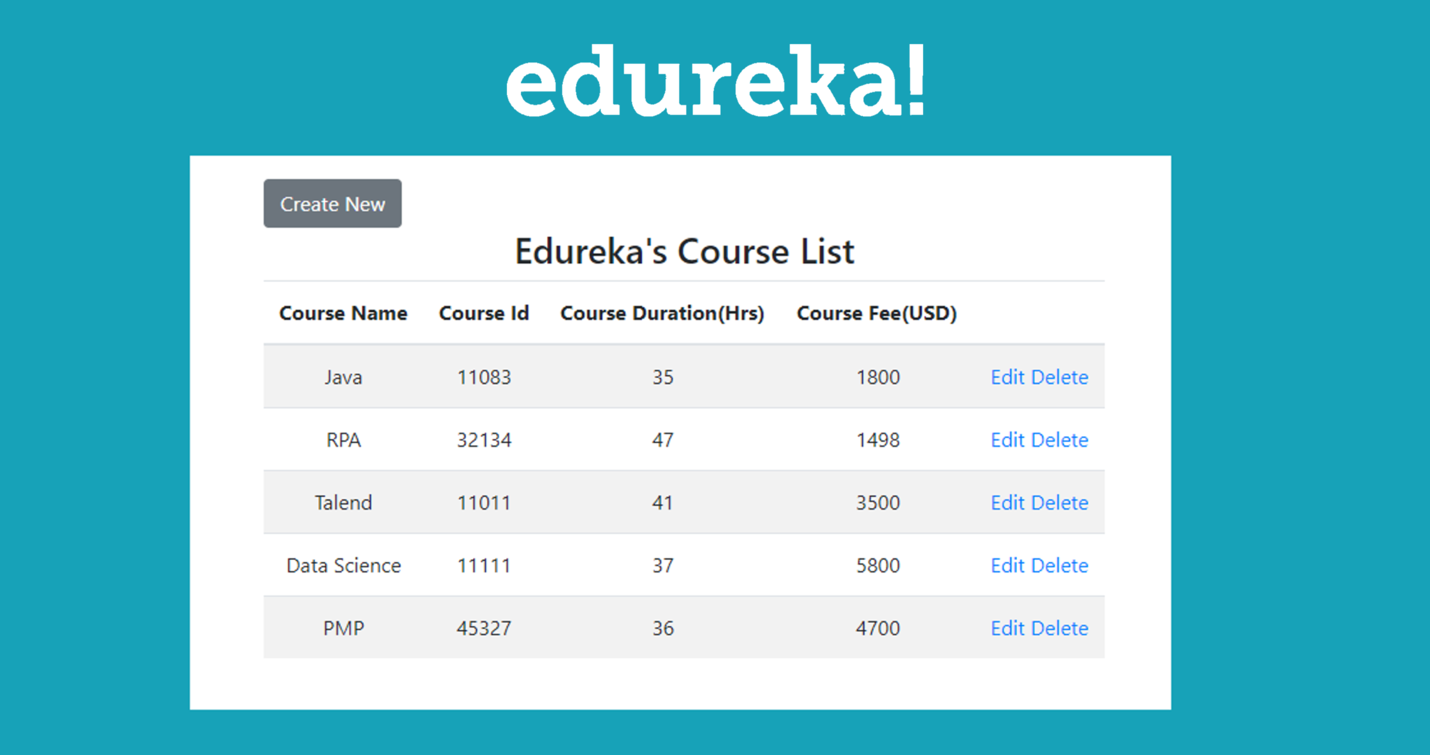
So, let me now show, how to build this view.

As you can see in the screenshot, the page will be containing four input fields and two buttons. One button will be submitting details filled in by the user into the database and the second button will display the complete list of courses available in the database. In order to make sure that the inputs are correct, you need to add some validations as well. Once done, you will be able to see this view using ‘/course’ URL. Below is the code, you will be needing to create **courseAddEdit.hbs**file:

**courseAddEdit.hbs**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | <!-- Obtaining value from the course controller -->  <h3>{{viewTitle}}</h3>  <form action="/course" method="POST">  <input type="hidden" name="\_id" value="{{course.\_id}}">  <div class = "form-group">  <label>Course Name</label>  <input type="text" maxlength="100" class="form-control" name="courseName" placeholder="Course Name" value="{{course.courseName}}">  <div class="text-danger">  {{course.courseNameError}}</div>  </div>  <div class = "form-group">  <label>Course ID</label>  <input type="number" min='10000' max='99999' class="form-control" name="courseId" placeholder="Course Id" value="{{course.courseId}} " required>  </div>  <div class = "form-row">  <div class = "form-group col-md-6">  <label>Course Duration</label>  <input type="number" min='10' max='99' class="form-control" name="courseDuration" placeholder="Course Duration (Hrs)" "{{course.courseDuration}} " required>  </div>  <div class = "form-group col-md-6">  <label>Course Fee</label>  <input type="number" min='100' max="100000" class="form-control" name="courseFee" placeholder="Course Fee (USD)" "{{course.courseFee}} " required>  </div>  </div>  <div class="form-group">  <button type="submit" class="btn btn-info"><i class="fa fa-database"></i> Submit</button>  <a class="btn btn-secondary" href="/course/list"><i class="fa fa-list-alt"></i> View All Courses</a>  </div>  </form> |

Now, let me show you the next view i.e **list.hbs**, which will retrieve the complete list of available courses from the database and display them on your screen:



**EduNet !**

**Edunet’s**

In this view, I am using a table to display the list of courses. This table will have five columns where the first four will display the course details while the last column will enable you to edit/delete a record directly from the application interface. The controllers of these functions have been already created in the **script.js** file. So, the only thing left is to add the view and in order to do so create a **list.hbs** file and type in the below-written code.

**list.hbs**

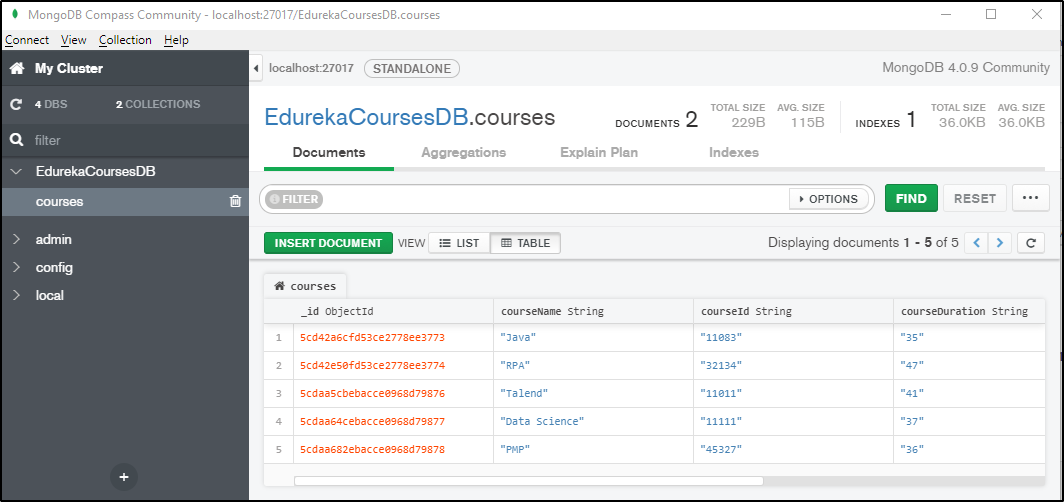
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <div>  <a class="btn btn-secondary" href="/course"><i class="fa fa-plus"></i> Create New</a>  <h3 align="center">Edunet's Course List</h3>  </div>  <table class="table table-striped">  <thead>  <tr>  <th>Course Name</th>  <th>Course Id</th>  <th>Course Duration(Hrs)</th>  <th>Course Fee(USD)</th>  <th></th>  </tr>  </thead>  <tbody>  {{#each list}}  <tr align="center">  <td>{{this.courseName}}</td>  <td>{{this.courseId}}</td>  <td>{{this.courseDuration}}</td>  <td>{{this.courseFee}}</td>  <td>  <a href="/course/{{this.\_id}}"> Edit </a>  <a href="/course/delete/{{this.\_id}}" onclick="return confirm('Are you sure to delete this record ?');"> Delete </a>  </td>  </tr>  {{/each}}  </tbody>  </table> |

This concludes are the coding part, now it’s time to test our application. For that, open the command prompt and navigate to the project folder or if you are using an IDE open the terminal and type in the below command to start the server.

|  |  |
| --- | --- |
| 1 | nodemon script.js |

Now you can launch your application in any browser at  **http://localhost:8080**.

Once you have added your own data, you can go back in MongoDB and check whether the data has been added there or not. If you refer the below screenshot, you will see all my data has been successfully added. Which means my MongoDB is connected and working perfectly with my Node.js API.



**Reference:** https://www.edureka.co/blog/node-js-mongodb-tutorial/