Coding Panel

Learn about coding and technology

NODE.JS MYSQL 0

Build a simple web application using Node.js and MySQL

By CodingPanel Editor - December 2, 2020

Time to Read: **13 min** - 2509 words

Facebook
Twitter
WhatsApp
LinkedIn Email
Like 2 Reddit

Table of Contents

In this article, we will build a basic To-do web application using **Node.js** and **MySQL** database. We will use the Express.js framework to keep

Privacy - Terms

our application short and simple. Moreover, we will use the Express Handlebars as our view engine.

A simple Todo app using Node.JS and MySQL

The To-do application will have the following functionalities:

- Add a task to the list.
- Remove a task.
- Update status (completed or ongoing) of a task.

We will incrementally build our web application to cover each of the topics thoroughly.

Installing the dependencies

First, let us create the folder for our application and install dependencies. As you might know, **Node.js** comes with npm (Node Package Manager), which will allow you to install the required modules and packages.

Let us now create a directory named todoApp and navigate to that directory.

Run the **npm** init command to initialize the project. You will have to answer some questions here regarding the detail of the project. If you want to skip them, write `npm init -y` instead. This will initialize the project and create a **package.json** file, which will contain project details and dependencies.

This is how my package.json file looks like:

Privacy - Terms

Now let’s install the required packages.

Express.js/Express handlebars

Install the **Express.js** and **express handlebars** using the following command.

```
npm install express express-handlebars --save
```

body-parser Middleware

Run the command given below to install the body-parser middleware.

```
npm install body-parser --save
```

MySQL module

Install the MySQL module

```
npm install mysql --save
```

nodemon tool

Install the nodemon tool to restart the server automatically whenever any changes are made.

```
npm install --save-dev nodemon
```

The ToDo application Basic Functionalities

Let's add the basic functionalities to our application. Create the **inde.js** file as shown below:

```
const express = require('express');
const hbs = require('express-handlebars');
const bodyParser = require("body-parser");

const app = express();


app.use(express.static('public'));

app.engine('hbs', hbs({
    layoutsDir: __dirname + '/views/layouts',
    defaultLayout: 'main',
    extname: '.hbs'
}));

app.set('view engine', 'hbs');

app.use(bodyParser.urlencoded({
    extended: true
}))

app.use(bodyParser.json())


app.get('/', (req, res) => {
    res.render('index', {
        items: []
    });
});


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

    console.log(req.body)
    res.redirect('/')
})

// port where app is served
app.listen(3000, () => {
    console.log('The web server has started on port 3000');
});
```

At the top of this file, we imported modules such as **Express.js** and **body-parser**, etc. Then, we created an express application in line 5. Moreover, to serve static files, for example, **CSS** files, we set the root directory to 'public' using the

Privacy - Terms

built-in middleware, which means that Express will look up in the public directory anytime you use a static file.

The Usage of HTML Templates

As already mentioned, we will use handlebars as our **HTML templates**. We need to configure and set the template engine, which is the code from line number 10 to 16. All our template files will be stored in the views folder and will have the extension `.hbs`. Moreover, we have added our **HTML** boilerplate in the main.hbs file that is stored in the layouts directory inside the views folder.

We use the body-parser module to parse the incoming request body and store it in `req.body` to easily access the data. At line numbers 18 and 20, we use middlewares to parse **JSON** and URL encoded data submitted through the **HTML** form using the **POST** method.
Then, we handle `get` and `post` requests using the `app.get()` method and the `app.post()` method. For now, the `app.get()` renders the index page, and the `app.post()` method displays the submitted data in the console and redirects back to the index page.
Now, let's go through the code of **main.hbs** and **index.hbs** templates.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Todo App</title>
      <link rel="stylesheet" type="text/css" href="/styles/index.css">
   </head>
   <body>
      {{{body}}}
   </body>
</html>
```

The above **main.hbs** file contains the **HTML** boilerplate. Here, we also add an external **CSS** file stored in the styles folder, which is inside the public folder.

Privacy - Terms

We add the body part to the **index.hbs** file stored in the views folder.

```
<div class="container">
   <h1>To Do App</h1>
   <h2>Enter your tasks here</h2>
   <form method="POST" action="/" >
      <input type="text" name="task" id="task" required>
      <button type="submit" id="submit">Add Task</button>
   </form>
   <div id="tasks_container">
      <ul>
         {{#each items}}
         <li>{{task}}</li>
         {{/each}}
      </ul>
   </div>
</div>
```

As you can see, the **index.hbs** file contains the form to insert the tasks. When the user presses the submit button, a post request will be sent to the server, and the given `app.post('/')` will get called. Moreover, we have a `<div>` that will display the tasks. Right now, we haven't passed anything there. Therefore, nothing will be shown.

Later in the article, we will send data by fetching it from the database.

Consider the index.css file for some basic styling of our app.

Styling the application

```
body {
    min-width: 60%;
    width: 80%;
    margin: auto;
}

.container {
    background-color: lightblue;
    padding: 2%
}

h1 {
```

Privacy - Terms

```
    color: navy;
}

#task {
    padding: 1% 2%;
    border-color: navy;
}

#submit {
    padding: 1% 2%;
    border-color: navy;
    background-color: navy;
    color: white;
}
```

Now Let's run the app by going to 127.0.0.1:3000 to test the code we wrote so far.

Now add enter a task in the textbox and click the button.

Privacy - Terms

Note: The task we just added is displayed in the console.

Connection to the MySQL Database

Let's now go a step further and create a connection to the database. For that, we create a models folder and place a **taskModel.js** file there.

Below is the code that we are going to use to connect to the MySQL database server.

```
var mysql = require('mysql');

var con = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: ""
});


con.connect(function (err) {
    if (err) throw err;
    console.log("Connected to the database!");
});
```

Privacy - Terms

```
module.exports = con;
```

In the above code, first, we require the MySQL module and store it in the MySQL variable. Then, we use the `mysql.createConnection()` method to initialize the connection. To connect the MySQL Databased server, we need the hostname, user, and password. Here, I’m using **XAMPP** and **phpMyAdmin** for the database. So, I have added its default user and password.

Now, we open the connection using the `con.connect()` method. Finally, we export the module to use it in the **index.js** file.

Add the following line to the **index.js** file and save the code.

```
const con = require('./models/taskModel')
```

After we run the application, We get the following output, which means that we have successfully established a connection to the database!

Create a Database

To execute MySQL statements, we have the `query()` method of the connection object. It has two parameters, i.e., a script to be executed and a callback function.

Let’s create a database.

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: ""
});


con.connect(function(err) {
```

Privacy - Terms

```
    if (err) throw err;
    console.log("Connected to the database!");
    let query ="CREATE DATABASE IF NOT EXISTS todo_db ";
    con.query(query, (err, result)=>{
        if (err) throw err;
        console.log(result)
    })
  });
  module.exports = con;
```

In the above code, we store the script of creating the **todo_db** database in the query variable and execute it by passing it to the `con.query()` method. The callback function has two parameters, `err` and `result`. If an error occurs while executing the query, it ill be thrown. Otherwise, we display the result of the executed query.

Create a Table

Now, let's create a table where we can store the tasks.

```
  var mysql = require('mysql');

  var con = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "",
    database: "todo_db"
  });


  con.connect(function(err) {
    if (err) throw err;
    console.log("Connected to the database!");
    let query ="CREATE TABLE IF NOT EXISTS Todo (task_id int NOT NULL
AUTO_INCREMENT, task VARCHAR(255) NOT NULL, status VARCHAR(255),
PRIMARY KEY (task_id))";
    con.query(query, (err, result)=>{
```

Privacy - Terms

```
        if (err) throw err;
        console.log(result)
    })
  });
  module.exports = con;
```

Since we have already created the database, we need to include it while initializing the connection, i.e., add database: "todo_db" to the object passed in the `mysql.createConnection()` method.

Our new table, Todo , as sown in the code above, contains three columns:

- **task_id**: It is the primary key and is incremented automatically.
- **task**: This field will contain the task inserted by the user.
- **status**: It will have the task's status, i.e., whether it is completed or not.

Add records to the table

Now that our database and table are created, let's write some code to insert some tasks into the table **Todo**.

```
const express = require('express');
const hbs = require('express-handlebars');
const bodyParser = require("body-parser");

const app = express();

const con = require('./models/taskModel')
app.use(express.static('public'));

app.engine('hbs', hbs({
    layoutsDir: __dirname + '/views/layouts',
    defaultLayout: 'main',
    extname: '.hbs'
}));

app.set('view engine', 'hbs');

app.use(bodyParser.urlencoded({
    extended: true
}))

app.use(bodyParser.json())
```

Privacy - Terms

```
app.get('/', (req, res) => {
    res.render('index');
});


app.post('/', (req, res) => {
    console.log(req.body)
    let query = "INSERT INTO Todo (task, status) VALUES ?";
    data = [
        [req.body.task, "ongoing"]
    ]
    con.query(query, [data], (err, result) => {
        if (err) throw err;
        console.log(result)
        res.redirect('/')
    })
})

// port where app is served
app.listen(3000, () => {
    console.log('The web server has started on port 3000');
});
```

In the `app.post()` method, we write a script to insert a task into the **Todo** table. As you can observe, there is a `?` after VALUES in the insert statement. This acts as a placeholder for the data to be inserted. Later, the data is passed as an argument to the `con.query()` method. Moreover, the data to be inserted is written as an array, i.e., `data = [[req.body.task, "ongoing"]]` a 2-D array that allows you to add multiple rows to the database.

Since the **task_id** column gets incremented automatically, we do not need to include it. We pass `req.body.task` for the task column and `ongoing` for the status, as default task.

Below is the output we get when we add a task.

Privacy - Terms

As we have inserted 1 row, the value of affectedRows is 1. The result object also contains the inserted with a value of 1, since it was the first item added.

Show the Tasks on the Application

Now we added the tasks, let's display them!

```
const express = require('express');
const hbs = require('express-handlebars');
const bodyParser = require("body-parser");

const app = express();

const con = require('./models/taskModel')
app.use(express.static('public'));

app.engine('hbs', hbs({
    helpers: {
        isCompleted: function (status) {
            if (status == "completed") {
                return true
            } else {
                return false
            }
        },
    },
    layoutsDir: __dirname + '/views/layouts',
    defaultLayout: 'main',
    extname: '.hbs'
}));

app.set('view engine', 'hbs');

app.use(bodyParser.urlencoded({
    extended: true
}))

app.use(bodyParser.json())


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

    let query = "SELECT * FROM Todo";
    let items = []
    con.query(query, (err, result) => {
        if (err) throw err;
        items = result
        console.log(items)
        res.render('index', {
            items: items
```

Privacy - Terms

```
            })
        })

    });


    app.post('/', (req, res) => {
        console.log(req.body)
        let query = "INSERT INTO Todo (task, status) VALUES ?";
        data = [
            [req.body.task, "ongoing"]
        ]
        con.query(query, [data], (err, result) => {
            if (err) throw err;
            console.log(result)
            res.redirect('/')
        })



    })

    // port where app is served
    app.listen(3000, () => {
        console.log('The web server has started on port 3000');
    });
```

In the `app.get('/')` method, we retrieve all the rows from the **Todo** and pass them to the **index.hbs** template. We set the class of the list item based on the status. For example, if its status is *ongoing*, then its class will be `ongoing` as well. To do that, we define a helper function `isCompleted()` when initializing handlebars. We use these classes to style the completed and ongoing tasks accordingly. The code of the **index.hbs** and the **index.css** files are given below.

HBS File

```
<div class="container">
   <h1>To Do App</h1>
   <h2>Enter your tasks here</h2>
   <form method="POST" action="/" >
      <input type="text" name="task" id="task" required>
      <button type="submit" id="submit">Add Task</button>
   </form>
   <div id="tasks_container">
      <ul>
         {{#each items}}
         {{#if (isCompleted status)}}
         <li class="completed">{{task}}</li>
         {{else}}
         <li class="ongoing">{{task}}</li>
         {{/if}}
```

Privacy - Terms

```
            {{/each}}
        </ul>
    </div>
</div>
```

CSS File

```
body {
    min-width: 60%;
    width: 80%;
    margin: auto;
}

.container {
    background-color: lightblue;
    padding: 2%
}

h1 {
    color: navy;
}

#task {
    padding: 2% 5%;
    border-color: navy;
}

#submit {
    padding: 2% 5%;
    border-color: navy;
    background-color: navy;
    color: white;
}

.ongoing {
    color: navy;
}

.completed {
    text-decoration: line-through;
    color: gray;
}
```

The following screenshot shows the index page after adding some tasks.

Privacy - Terms

Cool, right?

Update a Task

Let's now add the functionality to update the status, i.e., mark a task as *completed* or *ongoing*.

Add the following route to the **index.js** file.

```
app.get('/:status/:id', (req, res) => {
    console.log(req.params)
    let query = "UPDATE Todo SET status='" + req.params.status + "'
WHERE task_id=" + req.params.id
    con.query(query, (err, result) => {
        if (err) throw err;
        console.log(result)
        res.redirect('/')
    })

});
```

The updated **index.hbs** file is given below.

```
<div class="container">
   <h1>To Do App</h1>
   <h2>Enter your tasks here</h2>
   <form method="POST" action="/" >
      <input type="text" name="task" id="task" required>
      <button type="submit" id="submit">Add Task</button>
```

Privacy - Terms

```
      </form>
      <div id="tasks_container">
         <ul>
            {{#each items}}
            {{#if (isCompleted status)}}
            <li class="completed">{{task}}</li>
            <a href="/ongoing/{{task_id}}">Mark it as ongoing</a>
            {{else}}
            <li class="ongoing">{{task}}</li>
            <a href="/completed/{{task_id}}">Mark it as complete</a>
            {{/if}}
            {{/each}}
         </ul>
      </div>
   </div>
```

When the user clicks the **Mark it as complete** anchor tag (or **Mark it as ongoing** anchor tag), the new status code and the **task_id** get passed as URL parameters, and the corresponding route method defined in the **index.js** file runs, where we update the status of the specified task.

Let's mark the first task back to ongoing.

Delete a Task

Now, let's add the functionality to delete a task from the database table.

Add the following code in the **index.js** file.

```
app.get('/:id', (req, res) => {
    console.log(req.params)
    let query = "DELETE FROM Todo WHERE task_id=" + req.params.id
    con.query(query, (err, result) => {
        if (err) throw err;
        console.log(result)
        res.redirect('/')
    })

});
```

We added a delete icon in the **index.hbs** file. When the user presses it, the corresponding `task_id` gets passed as a URL parameter, and the above route method runs. In this method, we send a query to the database to delete the row with that `id`.

The updated **index.hbs** file is given below.

```
<div class="container">
   <h1>To Do App</h1>
   <h2>Enter your tasks here</h2>
```

Privacy - Terms

```
    <form method="POST" action="/" >
       <input type="text" name="task" id="task" required>
       <button type="submit" id="submit">Add Task</button>
    </form>
    <div id="tasks_container">
       <ul>
          {{#each items}}
          {{#if (isCompleted status)}}
          <li class="completed">{{task}}</li>
          <a href="/{{task_id}}"><i class="fa fa-trash"></i></a><a
href="/ongoing/{{task_id}}">Mark it as ongoing</a>
          {{else}}
          <li class="ongoing">{{task}}</li>
          <a href="/{{task_id}}"><i class="fa fa-trash"></i></a><a
href="/completed/{{task_id}}">Mark it as complete</a>
          {{/if}}
          {{/each}}
       </ul>
    </div>
 </div>
```

Add the following code in the `<head>` section of the main.hbs file to FontAwsome toolkit.

```
 <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/4.7.0/css/font-awesome.min.css">
```

The following screenshot shows the index page after we delete the 2nd item.

Privacy - Terms

Summary

- ✓ Create a project using the npm init command.
- ✓ Create an **index.js** file and write the route handling functionality there.
- ✓ Create the connection to the database using the `mysql.createConnection()` method.
- ✓ Open the connection using the `connect()` method of the connection object.
- ✓ Execute **MySQL** scripts using the `query()` method by passing a string containing the script.

That's it! You have successfully created a web application using Node.js, Express.js, and MySQL.

CodingPanel Editor

Blog Hero

Previous

Program to get yesterday's date in Java

Next

Print Prime Numbers from 1 to N in Java

Leave a Reply

Your email address will not be published. Required fields are marked *

COMMENT

NAME *

EMAIL *

WEBSITE

☐

SAVE MY NAME, EMAIL, AND WEBSITE IN THIS BROWSER FOR THE NEXT TIME I COMMENT.

POST COMMENT