TUTORIAL

How To Get Started with the MERN Stack

MongoDB Node.js JavaScript React

By Ojini Chizoba Jude

Published on December 12, 2019 41.8k

While this tutorial has content that we believe is of great benefit to our community, we have not yet tested or edited it to ensure you have an error-free learning experience. It's on our list, and we're working on it! You can help us out by using the "report an issue" button at the bottom of the tutorial.

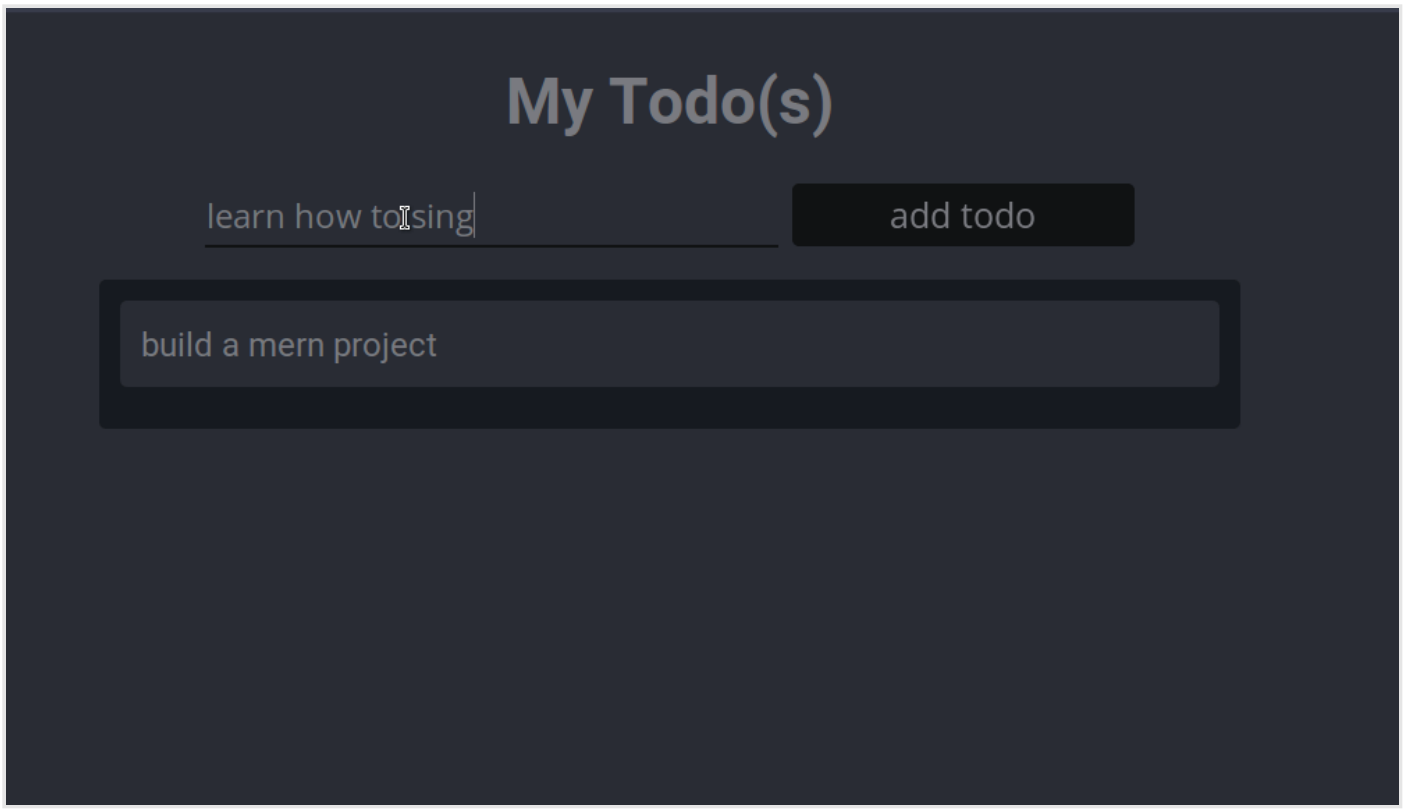
Introduction

The MERN stack consists of MongoDB, Express, React / Redux, and Node.js. The MERN stack is one of the most popular JavaScript stacks for building modern single-page web applications.

In this tutorial, you will build a **todo application** that uses a RESTful API, which you will also build later in this tutorial.

Prerequisites

To follow along with this tutorial, you will need to install node and npm on your computer. If you do not have node installed, follow the instructions on the nodejs website. You will also need a code editor that you are familiar with, preferably one that has support for JavaScript code highlighting.



Application Code Setup

Let's start with the setup. Open your terminal and create a new file directory in any convienient location on your local machine. You can name it anything but in this example it is called `Todo`.

```
$ mkdir Todo
```

Now enter into that file directory

```
$ cd Todo
```

The next step is to initialize the project with a `package.json` file. This file will contain some information about our app and the dependencies that it needs to run. You can use `npm init` and follow the instructions when prompted, or you can use `npm init -y` to use the default values.

Node Server Setup

To run our javascript code on the backend we need to spin up a server that will compile our code. The server can be created in two ways: first is to use the built in http module in node; second is to make use of the expressjs framework.

This tutorial will use expressjs. It is a nodejs HTTP framework that handles a lot of things out of the box and requires very little code to create fully functional RESTful APIs. To use express, install it using npm:

```
$ npm install express
```

Next, install the `dotenv` module, it allows you to load environment variables from a `.env` file into `process.env`:

```
$ npm install dotenv
```

Now create a file `index.js` and type the code below into it and save:

```
const express = require('express');
require('dotenv').config();

const app = express();

const port = process.env.PORT || 5000;

app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept
  next();
});

app.use((req, res, next) => {
  res.send('Welcome to Express');
});

app.listen(port, () => {
  console.log(`Server running on port ${port}`)
});
```

This snipped from the code above helps handle CORS related issues that you might face when trying to access the api from different domains during development and testing:

```
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept
  next();
});
```

It's time to start our server to see if it works. Open your terminal in the same directory as your index.js file and type:

```
$ node index.js
```

If every thing goes well, you should see **Server running on port 5000** in your terminal.

Routes

There are three things that the app needs to do: create a task; view all tasks; and delete a completed task. For each task, we need to create routes that will define various endpoints that the todo app will depend on. So let's create a folder `routes` and create a file `api.js` with the following code in it.

```
$ mkdir routes
```

Edit `api.js` and paste the following code in it:

```
const express = require ('express');
const router = express.Router();

router.get('/todos', (req, res, next) => {

});

router.post('/todos', (req, res, next) => {

});
```

```
router.delete('/todos/:id', (req, res, next) => {

})

module.exports = router;
```

Models

Now comes the interesting part, since the app is going to make use of mongodb which is a noSql database, we need to create a model and a schema. Models are defined using the Schema interface. The Schema allows you to define the fields stored in each document along with their validation requirements and default values. In essence the Schema is a blueprint of how the database will be constructed. In addition, you can define static and instance helper methods to make it easier to work with your data types, and also virtual properties that you can use like any other field, but which aren't actually stored in the database.

To create a Schema and a model, install mongoose which is a node package that makes working with mongodb easier.

```
$ npm install mongoose
```

Create a new folder in your root directory and name it `models`. Inside it create a file and name it `todo.js` with the following code in it:

```
$ mkdir models
```

Paste the following into `todo.js` with your text editor:

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

//create schema for todo
const TodoSchema = new Schema({
  action: {
    type: String,
    required: [true, 'The todo text field is required']
  }
```

```
})

//create model for todo
const Todo = mongoose.model('todo', TodoSchema);

module.exports = Todo;
```

Now we need to update our routes to make use of the new model.

```
const express = require ('express');
const router = express.Router();
const Todo = require('../models/todo');

router.get('/todos', (req, res, next) => {

  //this will return all the data, exposing only the id and action field to the client
  Todo.find({}, 'action')
    .then(data => res.json(data))
    .catch(next)
});

router.post('/todos', (req, res, next) => {
  if(req.body.action){
    Todo.create(req.body)
      .then(data => res.json(data))
      .catch(next)
  }else {
    res.json({
      error: "The input field is empty"
    })
  }
});

router.delete('/todos/:id', (req, res, next) => {
  Todo.findOneAndDelete({"_id": req.params.id})
    .then(data => res.json(data))
    .catch(next)
})

module.exports = router;
```

Database

We need a database where we will store our data. For this we will make use of mlab. Follow this doc to get started with mlab. After setting up your database you need t

file with the following code:

```
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const routes = require('./routes/api');
const path = require('path');
require('dotenv').config();

const app = express();

const port = process.env.PORT || 5000;

//connect to the database
mongoose.connect(process.env.DB, { useNewUrlParser: true })
  .then(() => console.log(`Database connected successfully`))
  .catch(err => console.log(err));

//since mongoose promise is depreciated, we overide it with node's promise
mongoose.Promise = global.Promise;

app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept
  next();
});

app.use(bodyParser.json());

app.use('/api', routes);

app.use((err, req, res, next) => {
  console.log(err);
  next();
});

app.listen(port, () => {
  console.log(`Server running on port ${port}`)
});
```

In the code above we made use of `process.env` to access environment variables, which need to be created. Create a file in your root directory with name `.env` and edit:

> DB = 'mongodb://<USER\>:<PASSWORD\>@ds039950.mlab.com:39950/todo'

Make sure you use your own mongodb URL from mlab after you created your database and user. Replace `<USER>` with the username and `<PASSWORD>` with the password of the user you created. To work with environment variable we have to install a node package called `dotenv` that makes sure we have access to environment variable stored in the `.env` file.

```
$ npm install dotenv
```

Then require and configure it in `index.js`:

```
require('dotenv').config()
```

Using environment variables instead of writing credentials to the application code directly can hide sensitive information from our versioning system. It is considered a best practice to separate configuration and secret data from application code in this manner.

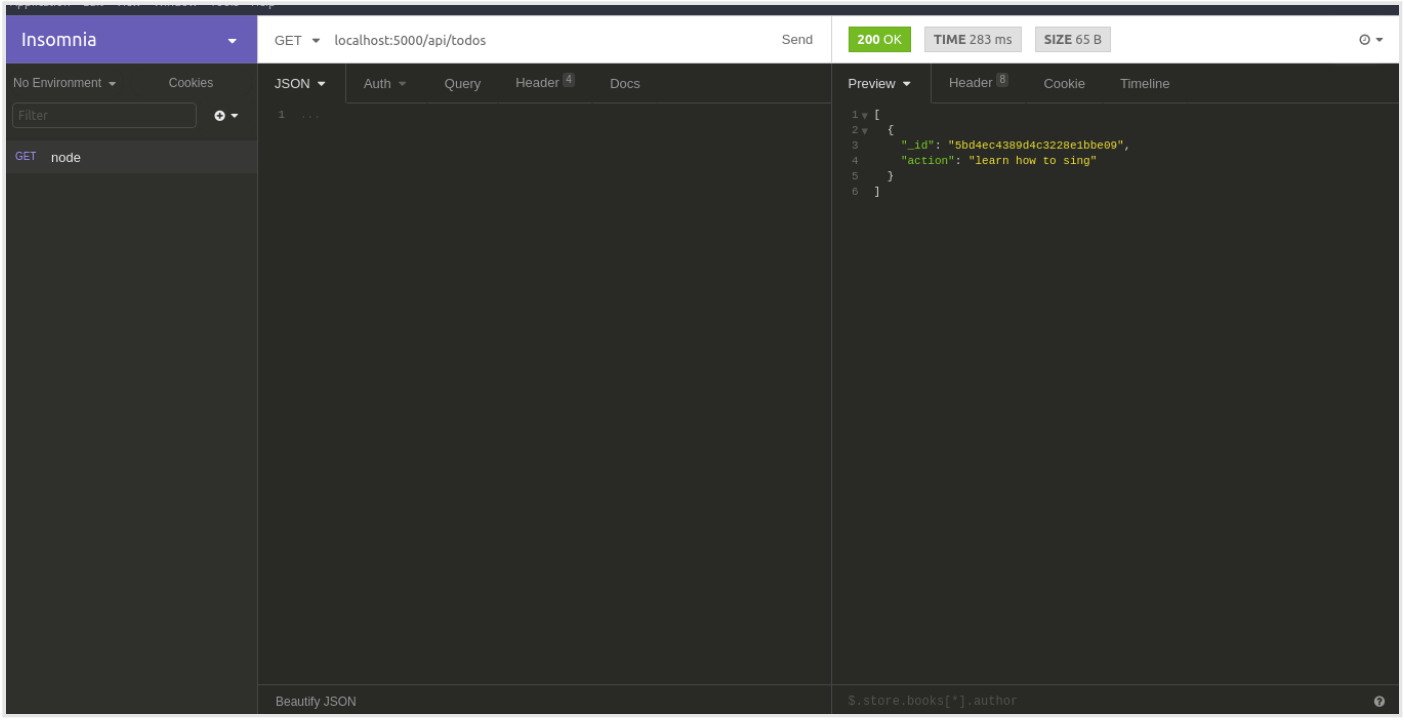
Testing Api

This is the part we start trying out things to make sure our RESTful api is working. Since our frontend is not ready yet, we can make use of some api development clients to test our code. You can use of Postman or Insomnia or if there is another client that you prefer for testing APIs, you can use of it.

Start your server using the command:

```
$ node index.js
```

Now open your client, create a get method and navigate to http://localhost:5000/api/todos



You should test all the api endpoints and make sure they are working. For the endpoints that require body, you should send json back with the necessary fields since it's what we setup in our code.

Creating the Frontend

Since we are done with the functionality we want from our api, it is time to create an interface for the client to interact with the api. To start out with the frontend of the todo app, we will use the `create-react-app` command to scaffold our app.

In the same root directory as your backend code, which is the todo directory, run:

```
$ create-react-app client
```

This will create a new folder in your `todo` directory called `client`, where you will add all the react code.

Running the React App

Before testing the react app, there are a number od dependencies that nee

1. Install **concurrently** as a dev dependency:

```
$ npm install concurrently --save-dev
```

Concurrently is used to run more than one command simultaneously from the same terminal window.

1. Install nodemon as a dev dependency:

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

Nodemon is used to run the server and monitor it as well. If there is any change in the server code, nodemon will restart it automatically with the new changes.

1. Open your package.json file in the root folder of the app project, and paste the following code

```
{
"name": "todo",
"version": "1.0.0",
"description": "building todo app using mongodb, express, react and nodejs",
"main": "index.js",
"scripts": {
"start": "node index.js",
"start-watch": "nodemon index.js",
"dev": "concurrently \"yarn run start-watch\" \"cd client && yarn start\""
},
"author": "Ojini Jude",
"license": "MIT",
"dependencies": {
"body-parser": "^1.18.3",
"dotenv": "^6.1.0",
"express": "^4.16.4",
"mongoose": "^5.3.6"
},
"devDependencies": {
"concurrently": "^4.0.1",
"nodemon": "^1.18.4"
}
}
```

1. Enter into the client folder, then locate the package.json file and add the key value pair below inside it. This proxy setup in our package.json file will enable us make api calls without having to type the full url, just /api/todos will get all our todos

```
"proxy": "http://localhost:5000"
```

Open your terminal and run `npm run dev` and make sure you are in the todo directory and not in the client directory. Your app should be open and running on localhost:3000.

Creating your React Components

One of the advantages of react is that it makes use of components, which are reusable and also makes code modular. For our todo app, there will be two state components and one stateless component. Inside your `src` folder create another folder called `components` and inside it create three files `Input.js`, `ListTodo.js` and `Todo.js`.

Open `Input.js` file and paste the following

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


class Input extends Component {

  state = {
    action: ""
  }

  addTodo = () => {
    const task = {action: this.state.action}

    if(task.action && task.action.length > 0){
      axios.post('/api/todos', task)
        .then(res => {
          if(res.data){
            this.props.getTodos();
            this.setState({action: ""})
          }
        })
        .catch(err => console.log(err))
    }else {
      console.log('input field required')
```

```
    }
  }

  handleChange = (e) => {
    this.setState({
      action: e.target.value
    })
  }

  render() {
    let { action } = this.state;
    return (
      <div>
        <input type="text" onChange={this.handleChange} value={action} />
        <button onClick={this.addTodo}>add todo</button>
      </div>
    )
  }
}

export default Input
```

To make use of axios, which is a Promise based HTTP client for the browser and node.js, you need to cd into your client from your terminal and run yarn add axios or npm install axios

```
$ cd client
$ npm install axios
```

After that open your `ListTodo.js` file and paste the following code

```
import React from 'react';

const ListTodo = ({ todos, deleteTodo }) => {

  return (
    <ul>
      {
        todos &&
          todos.length > 0 ?
            (
              todos.map(todo => {
                return (
                  <li key={todo._id} onClick={() => deleteTodo(todo._id)}>{todo.action}</li>
                )
              })
            )
```

```
            :
            (
              <li>No todo(s) left</li>
            )
      }
    </ul>
  )
}

export default ListTodo
```

Then in your `Todo.js` file you write the following code

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

import Input from './Input';
import ListTodo from './ListTodo';

class Todo extends Component {

  state = {
    todos: []
  }

  componentDidMount(){
    this.getTodos();
  }

  getTodos = () => {
    axios.get('/api/todos')
      .then(res => {
        if(res.data){
          this.setState({
            todos: res.data
          })
        }
      })
      .catch(err => console.log(err))
  }

  deleteTodo = (id) => {

    axios.delete(`/api/todos/${id}`)
      .then(res => {
        if(res.data){
          this.getTodos()
        }
```

```
      })
      .catch(err => console.log(err))
  }

  render() {
    let { todos } = this.state;

    return(
      <div>
        <h1>My Todo(s)</h1>
        <Input getTodos={this.getTodos}/>
        <ListTodo todos={todos} deleteTodo={this.deleteTodo}/>
      </div>
    )
  }
}

export default Todo;
```

We need to make little adjustment to our react code. Delete the logo and adjust our `App.js` to look like this.

```
import React from 'react';

import Todo from './components/Todo';
import './App.css';

const App = () => {
  return (
    <div className="App">
      <Todo />
    </div>
  );
}

export default App;
```

Then paste the following code into `App.css`:

```
.App {
  text-align: center;
  font-size: calc(10px + 2vmin);
  width: 60%;
  margin-left: auto;
  margin-right: auto;
}
```

```css
input {
  height: 40px;
  width: 50%;
  border: none;
  border-bottom: 2px #101113 solid;
  background: none;
  font-size: 1.5rem;
  color: #787a80;
}

input:focus {
  outline: none;
}

button {
  width: 25%;
  height: 45px;
  border: none;
  margin-left: 10px;
  font-size: 25px;
  background: #101113;
  border-radius: 5px;
  color: #787a80;
  cursor: pointer;
}

button:focus {
  outline: none;
}

ul {
  list-style: none;
  text-align: left;
  padding: 15px;
  background: #171a1f;
  border-radius: 5px;
}

li {
  padding: 15px;
  font-size: 1.5rem;
  margin-bottom: 15px;
  background: #282c34;
  border-radius: 5px;
  overflow-wrap: break-word;
  cursor: pointer;
}

@media only screen and (min-width: 300px) {
  .App {
```

```
    width: 80%;
  }

  input {
    width: 100%
  }

  button {
    width: 100%;
    margin-top: 15px;
    margin-left: 0;
  }
}

@media only screen and (min-width: 640px) {
  .App {
    width: 60%;
  }

  input {
    width: 50%;
  }

  button {
    width: 30%;
    margin-left: 10px;
    margin-top: 0;
  }
}
```

Also in `index.css` add the following rules:

```
body {
  margin: 0;
  padding: 0;
  font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", "Roboto", "Oxygen",
    "Ubuntu", "Cantarell", "Fira Sans", "Droid Sans", "Helvetica Neue",
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  box-sizing: border-box;
  background-color: #282c34;
  color: #787a80;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, "Courier New",
    monospace;
}
```

Assuming no errors when saving all these files, the todo app should be ready and fully functional with the functionality discussed earlier: creating a task, deleting a task and viewing all your tasks.

Conclusion

In this tutorial, you created a todo app using the MERN stack. You wrote a frontend application using React that communicates with a backend application written using expressjs. You also created a Mongodb backend for storing tasks in a database.

About the authors

Ojini Chizoba Jude

is a Community author on DigitalOcean.

Still looking for an answer?