Getting Started with MERN

Part 1

In this tutorial series we're going to explore the MERN stack by building a real-world application from start to finish. The MERN stack consists of the following technologies:

Node.js: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js brings JavaScript to the server
MongoDB: A document-based open source database
Express: A Fast, unopinionated, minimalist web framework for Node.js
React: A JavaScript front-end library for building user interfaces

The MERN stack is very similar to the popular MEAN stack. The only difference here is that the MEAN stack is making use of Angular to build the front-end web application and the MERN stack is using React instead.
The application we'll be building in this tutorial series is a simple To-Do application. By using this example it's possible to demonstrate how to build a CRUD (Create, Read, Update, and Delete) application from scratch by using the MERN stack!

In this first part of this series we're going to complete the setup the React project for building the front-end part of the MERN stack sample application. In the next part we're going to continue with implementing the Node.js / Express server.

Setting Up The React Application

For this tutorial it is assumed that Node.js is installed on your system. If that is not the case please go to https://nodejs.org/ first and follow the installation instructions for your platform.

You can check if Node.js is installed on your system by typing in:

```
$ node -v
```

on the command line. This will print out the Node.js version which is installed on your system.
In the next step we're creating the initial React project by using the create-react-app script. What's great about create-react-app is that this script can be executed by using the npx command without the need to install it first on your system. Just execute the following command:

```
$ npx create-react-app mern-todo-app
```

Executing this command creates a new project directory mern-todo-app. Inside this folder you'll find the default React project template with all dependencies installed. Change into the newly created folder:

```
$ cd mern-todo-app
```

and start the development web server by running the following command:

```
$ npm start
```

Adding Bootstrap To The React Project

Next, we need to add the Bootstrap framework to our project. This is needed because we'll be making use of Bootstrap's CSS classes to build our user interface. Inside the project folder execute the following command to add the library:

```
$ npm install bootstrap
```

Next you need to make sure that Bootstrap’s CSS file is imported in App.js by adding the following line of code:

```
import "bootstrap/dist/css/bootstrap.min.css";
```

Furthermore you need to get rid of most of the default code which is contained in App.js, so that only the following code remains:

```
import React, { Component } from "react";
import "bootstrap/dist/css/bootstrap.min.css";
class App extends Component {
  render() {
    return (

        <div className="container">
          <h2>MERN-Stack Todo App</h2>
        </div>

    );
  }
}
export default App;
```

Setting Up React Router

The next thing we needs to be added to the project is the React Router package: react-router-dom:

```
$ npm install react-router-dom
```

With this package installed we’re ready to add the routing configuration in App.js. First of all the following import statement needs to be added:

```
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
```

Next, let’s embed the JSX code in a <Router></Router> element:

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
class App extends Component {
  render() {
    return (
      <Router>
        <div className="container">
          <h2>MERN-Stack Todo App</h2>
        </div>
      </Router>
    );
  }
}
export default App;
```

Inside the <Router> element we're now ready to add the router configuration inside that element:

```
<Route path="/" exact component={TodosList} />
<Route path="/edit/:id" component={EditTodo} />
<Route path="/create" component={CreateTodo} />
```

For each route which needs to be added to the application a new <Route> element is added. The attributes path and component are used to add the configuration settings for each route. By using the attribute path the routing path is set and by using the component attribute the path is connected with a component.

As you can see we need to add three routes to our application:

/

/create
/edit/:id
For these three routes we want to connect to three components:
TodosList
EditTodo
CreateTodo

Creating Components

To create the needed components in our application let's first create a new directory src/components and create three new files:

todos-list.component.js
edit-todo.component.js
create-todo.component.js

Let's add a basic React component implementation for each of those components:

todos-list.component.js:
```
import React, { Component } from 'react';
export default class TodosList extends Component {
   render() {
      return (
         <div>
            <p>Welcome to Todos List Component!!</p>
         </div>
      )
   }
}
```
edit-todo.component.js:
```
import React, { Component } from 'react';
export default class EditTodo extends Component {
   render() {
      return (
         <div>
            <p>Welcome to Edit Todo Component!!</p>
         </div>
      )
   }
}
```
create-todo.component.js:
```
import React, { Component } from 'react';
export default class CreateTodo extends Component {
   render() {
      return (
         <div>
            <p>Welcome to Create Todo Component!!</p>
         </div>
      )
   }
}
```

Creating The Basic Layout & Navigation

Next let's add a basic layout and navigation menu to our application. Because we've added the Bootstrap library before we can now make use of Bootstrap's CSS classes to implement the user interface of our web application. Extend the code in App.js to the following:

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import CreateTodo from "./components/create-todo.component";
import EditTodo from "./components/edit-todo.component";
import TodosList from "./components/todos-list.component";
import logo from "./logo.png";
class App extends Component {
  render() {
    return (
      <Router>
        <div className="container">
          <nav className="navbar navbar-expand-lg navbar-light bg-light">
            <a class="navbar-brand" href="https://codingthesmartway.com" target="_blank">
              <img src={logo} width="30" height="30" alt="CodingTheSmartWay.com" />
            </a>
            <Link to="/" className="navbar-brand">MERN-Stack Todo App</Link>
            <div className="collpase navbar-collapse">
              <ul className="navbar-nav mr-auto">
                <li className="navbar-item">
                  <Link to="/" className="nav-link">Todos</Link>
                </li>
                <li className="navbar-item">
                  <Link to="/create" className="nav-link">Create Todo</Link>
                </li>
              </ul>
            </div>
          </nav>
          <br/>
          <Route path="/" exact component={TodosList} />
          <Route path="/edit/:id" component={EditTodo} />
          <Route path="/create" component={CreateTodo} />
        </div>
      </Router>
    );
  }
}
export default App;
```

Let's check again what we can see in the Browser:

The navigation bar is displayed with two menu items included (Todos and Create Todo). By default the output of TodosList component is shown because it was connected to the default route of the application.
Clicking on the Create Todo link shows the output of CreateTodo component:

Implementing Create Todo Component
Let's add the CreateTodo component implementation in the next step in file create-todo.component.js. First we start by adding a constructor to the component class:

```
constructor(props) {
        super(props);
        this.state = {
            todo_description: '',
            todo_responsible: '',
            todo_priority: '',
            todo_completed: false
        }
    }
```
Inside the constructor we're setting the initial state of the component by assigned an object to this.state. The state comprises the following properties:
todo_description
todo_responsible
todo_priority
todo_completed
Furthermore we need to add methods which can be used to update the state properties:
```
onChangeTodoDescription(e) {
        this.setState({
            todo_description: e.target.value
        });
    }
    onChangeTodoResponsible(e) {
        this.setState({
            todo_responsible: e.target.value
        });
    }
    onChangeTodoPriority(e) {
        this.setState({
            todo_priority: e.target.value
        });
    }
```
Finally another method is needed to handle the submit event of the form which will be implemented to create a new todo item:
```
onSubmit(e) {
        e.preventDefault();

        console.log(`Form submitted:`);
        console.log(`Todo Description: ${this.state.todo_description}`);
        console.log(`Todo Responsible: ${this.state.todo_responsible}`);
        console.log(`Todo Priority: ${this.state.todo_priority}`);

        this.setState({
            todo_description: '',
            todo_responsible: '',
            todo_priority: '',
            todo_completed: false
        })
    }
```
Inside this method we need to call e.preventDefault to ensure that the default HTML form submit behaviour is prevented. Because the back-end of our application is not implemented yet we're only printing out what's currently available in the local component's state to the console. Finally we're making sure that the form is resetted by setting the resetting the state object.
Because in the four implemented methods we're dealing with the component's state object we need to make sure to bind those methods to this by adding the following lines of code to the constructor:
```
this.onChangeTodoDescription = this.onChangeTodoDescription.bind(this);
        this.onChangeTodoResponsible = this.onChangeTodoResponsible.bind(this);
        this.onChangeTodoPriority = this.onChangeTodoPriority.bind(this);
        this.onSubmit = this.onSubmit.bind(this);
```
Finally we need to add the JSX code which is needed to display the form:
```
render() {
```

```
return (
    <div style={{marginTop: 10}}>
        <h3>Create New Todo</h3>
        <form onSubmit={this.onSubmit}>
            <div className="form-group">
                <label>Description: </label>
                <input  type="text"
                        className="form-control"
                        value={this.state.todo_description}
                        onChange={this.onChangeTodoDescription}
                        />
            </div>
            <div className="form-group">
                <label>Responsible: </label>
                <input
                        type="text"
                        className="form-control"
                        value={this.state.todo_responsible}
                        onChange={this.onChangeTodoResponsible}
                        />
            </div>
            <div className="form-group">
                <div className="form-check form-check-inline">
                    <input  className="form-check-input"
                            type="radio"
                            name="priorityOptions"
                            id="priorityLow"
                            value="Low"
                            checked={this.state.todo_priority==='Low'}
                            onChange={this.onChangeTodoPriority}
                            />
                    <label className="form-check-label">Low</label>
                </div>
                <div className="form-check form-check-inline">
                    <input  className="form-check-input"
                            type="radio"
                            name="priorityOptions"
                            id="priorityMedium"
                            value="Medium"
                            checked={this.state.todo_priority==='Medium'}
                            onChange={this.onChangeTodoPriority}
                            />
                    <label className="form-check-label">Medium</label>
                </div>
                <div className="form-check form-check-inline">
                    <input  className="form-check-input"
                            type="radio"
                            name="priorityOptions"
                            id="priorityHigh"
                            value="High"
                            checked={this.state.todo_priority==='High'}
                            onChange={this.onChangeTodoPriority}
                            />
                    <label className="form-check-label">High</label>
                </div>
            </div>
```

```
                <div className="form-group">
                    <input type="submit" value="Create Todo" className="btn btn-primary" />
                </div>
            </form>
        </div>
    )
}
```

In the following you can see the complete source code which should now be available in create-todo.component.js:

```
import React, { Component } from 'react';
export default class CreateTodo extends Component {
    constructor(props) {
        super(props);
        this.onChangeTodoDescription = this.onChangeTodoDescription.bind(this);
        this.onChangeTodoResponsible = this.onChangeTodoResponsible.bind(this);
        this.onChangeTodoPriority = this.onChangeTodoPriority.bind(this);
        this.onSubmit = this.onSubmit.bind(this);
        this.state = {
            todo_description: '',
            todo_responsible: '',
            todo_priority: '',
            todo_completed: false
        }
    }
    onChangeTodoDescription(e) {
        this.setState({
            todo_description: e.target.value
        });
    }
    onChangeTodoResponsible(e) {
        this.setState({
            todo_responsible: e.target.value
        });
    }
    onChangeTodoPriority(e) {
        this.setState({
            todo_priority: e.target.value
        });
    }
    onSubmit(e) {
        e.preventDefault();

        console.log(`Form submitted:`);
        console.log(`Todo Description: ${this.state.todo_description}`);
        console.log(`Todo Responsible: ${this.state.todo_responsible}`);
        console.log(`Todo Priority: ${this.state.todo_priority}`);

        this.setState({
            todo_description: '',
            todo_responsible: '',
            todo_priority: '',
            todo_completed: false
        })
    }
    render() {
        return (
<div style={{marginTop: 10}}>
   <h3>Create New Todo</h3>
   <form onSubmit={this.onSubmit}>
      <div className="form-group">
         <label>Description: </label>
         <input  type="text"
               className="form-control"
               value={this.state.todo_description}
               onChange={this.onChangeTodoDescription}
               />
      </div>
      <div className="form-group">
         <label>Responsible: </label>
         <input
               type="text"
               className="form-control"
               value={this.state.todo_responsible}
               onChange={this.onChangeTodoResponsible}
               />
      </div>
      <div className="form-group">
         <div className="form-check form-check-inline">
            <input  className="form-check-input"
                  type="radio"
                  name="priorityOptions"
                  id="priorityLow"
                  value="Low"
                  checked={this.state.todo_priority==='Low'}
                  onChange={this.onChangeTodoPriority}
                  />
            <label className="form-check-label">Low</label>
         </div>
         <div className="form-check form-check-inline">
            <input  className="form-check-input"
                  type="radio"
                  name="priorityOptions"
                  id="priorityMedium"
                  value="Medium"
                  checked={this.state.todo_priority==='Medium'}
                  onChange={this.onChangeTodoPriority}
                  />
            <label className="form-check-label">Medium</label>
         </div>
         <div className="form-check form-check-inline">
            <input  className="form-check-input"
                  type="radio"
                  name="priorityOptions"
                  id="priorityHigh"
                  value="High"
                  checked={this.state.todo_priority==='High'}
                  onChange={this.onChangeTodoPriority}
                  />
            <label className="form-check-label">High</label>
         </div>
      </div>
      <div className="form-group">
```

```
                    <input type="submit" value="Create Todo" className="btn btn-primary" />
                </div>
            </form>
        </div>
    )
  }
}
```
The result which is delivered in the browser when accessing the /create path should correspond to what you can see in the following:

Filling out the form and submitting it by clicking on button Create Todo will output the data which have been entered on the console.

Part 2

The back-end will comprise HTTP endpoints to cover the following use cases:

- Retrieve the complete list of available todo items by sending an HTTP GET request
- Retrieve a specific todo item by sending HTTP GET request and provide the specific todo ID in addition
- Create a new todo item in the database by sending an HTTP POST request
- Update an existing todo item in the database by sending an HTTP POST request

Initiating The Back-end Project

To initiate the back-end project let's create a new empty project folder:

```
$ mkdir backend
```

Change into that newly created folder by using:

```
$ cd backend
```

Let's create a package.json file inside that folder by using the following command:

```
$ npm init -y
```

With the package.json file available in the project folder we're ready to add some dependencies to the project:

```
$ npm install express body-parser cors mongoose
```

Let's take a quick look at the four packages:

express: Express is a fast and lightweight web framework for Node.js. Express is an essential part of the MERN stack.
body-parser: Node.js body parsing middleware.
cors: CORS is a node.js package for providing an Express middleware that can be used to enable CORS with various options. Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.
mongoose: A Node.js framework which lets us access MongoDB in an object-oriented way.
Finally we need to make sure to install a global package by executing the following command:

```
$ npm install -g nodemon
```

Nodemon is a utility that will monitor for any changes in your source and automatically restart your server. We'll use nodemon when running our Node.js server in the next steps.

Inside of the backend project folder create a new file named server.js and insert the following basic Node.js / Express server implementation:

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const cors = require('cors');
const PORT = 4000;

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

app.listen(PORT, function() {
   console.log("Server is running on Port: " + PORT);
});
```

With this code we're creating an Express server, attaching the cors and body-parser middleware and making the server listening on port 4000.

Start the server by using nodemon:

```
$ nodemon server
```

You should now see an output similar to the following:

As we're able to see the out Server is running on Port: 4000 we know that the server has been started up successfully and is listing on port 4000.

Installing MondoDB
Now that we've managed to set up a basic Node.js / Express server we're ready to continue with the next task: setting up the MongoDB database.

First of all we need to make sure that MongoDB is installed on your system. On MacOS this task can be completed by using the following command:

```
$ brew install mongodb
```

If you're working on Windows or Linux follow the installation instructions from https://docs.mongodb.com/manual/administration/install-community/.

Having installed MongoDB on your system you need to create a data directory which is used by MongoDB:

```
$ mkdir -p /data/db
```

Before running mongod for the first time, ensure that the user account running mongod has read and write permissions for the directory.

Now we're ready to start up MongoDB by executing the following command:

```
$ mongod
```

Executing this command will give you the following output on the command line:

This shows that the database is now running on port 27017 and is waiting to accept client connections.

Creating A New MongoDB Database

The next step is to create the MongoDB database instance. Therefore we’re connecting to the database server by using the MondoDB client on the command line:

```
$ mongo
```

Once the client is started it prompts you to enter database commands. By using the following command we’re creating a new database with the name todos:

```
use todos
```

Connecting To MongoDB By Using Mongoose

Let’s return to the Node.js / Express server implementation in server.js. With the MongoDB database server running we’re now ready to connect to MongoDB from our server program by using the Mongoose library. Change the implementation in server.js to the following:

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const cors = require('cors');
const mongoose = require('mongoose');
const PORT = 4000;

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

mongoose.connect('mongodb://127.0.0.1:27017/todos', { useNewUrlParser: true });
const connection = mongoose.connection;

connection.once('open', function() {
    console.log("MongoDB database connection established successfully");
})

app.listen(PORT, function() {
    console.log("Server is running on Port: " + PORT);
});
```

On the console you should now also see the output MongoDB database connection established successfully in addition.

Create a Mongoose Schema

By using Mongoose we’re able to access the MongoDB database in an object-oriented way. This means that we need to add a Mongoose schema for our Todo entity to our project implementation next.

Inside the back-end project folder create a new file todo.model.js and insert the following lines of code to create a Todo schema:

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

let Todo = new Schema({
   todo_description: {
      type: String
   },
   todo_responsible: {
      type: String
   },
   todo_priority: {
      type: String
   },
   todo_completed: {
      type: Boolean
   }
});

module.exports = mongoose.model('Todo', Todo);
```

With this code in place we're now ready to access the MongoDB database by using the Todo schema.

Implementing The Server Endpoints

In the last step let's complete the server implementation in server.js by using the Todo schema we've just added to implement the API endpoints we'd like to provide.

To setup the endpoints we need to create an instance of the Express Router by adding the following line of code:

```
const todoRoutes = express.Router();
```

The router will be added as a middleware and will take control of request starting with path /todos:

```
app.use('/todos', todoRoutes);
```

First of all we need to add an endpoint which is delivering all available todos items:

```
todoRoutes.route('/').get(function(req, res) {
   Todo.find(function(err, todos) {
      if (err) {
         console.log(err);
      } else {
         res.json(todos);
      }
   });
});
```

The function which is passed into the call of the method get is used to handle incoming HTTP GET request on the /todos/ URL path. In this case we're calling Todos.find to retrieve a list of all todo items from the MongoDB database. Again the call of the find methods takes one argument: a callback function which is executed once the result is available. Here we're making sure that the results (available in todos) are added in JSON format to the response body by calling res.json(todos).

The next endpoint which needs to be implemented is /:id. This path extension is used to retrieve a todo item by providing an ID. The implementation logic is straight forward:

```
todoRoutes.route('/:id').get(function(req, res) {
    let id = req.params.id;
    Todo.findById(id, function(err, todo) {
        res.json(todo);
    });
});
```

Here we're accepting the URL parameter id which can be accessed via req.params.id. This id is passed into the call of Tood.findById to retrieve an issue item based on it's ID. Once the todo object is available it is attached to the HTTP response in JSON format.

Next, let's add the route which is needed to be able to add new todo items by sending a HTTP post request (/add):

```
todoRoutes.route('/add').post(function(req, res) {
    let todo = new Todo(req.body);
    todo.save()
        .then(todo => {
            res.status(200).json({'todo': 'todo added successfully'});
        })
        .catch(err => {
            res.status(400).send('adding new todo failed');
        });
});
```

The new todo item is part the the HTTP POST request body, so that we're able to access it view req.body and therewith create a new instance of Todo. This new item is then saved to the database by calling the save method.

Finally a HTTP POST route /update/:id is added:

```
todoRoutes.route('/update/:id').post(function(req, res) {
    Todo.findById(req.params.id, function(err, todo) {
        if (!todo)
            res.status(404).send("data is not found");
        else
            todo.todo_description = req.body.todo_description;
            todo.todo_responsible = req.body.todo_responsible;
            todo.todo_priority = req.body.todo_priority;
            todo.todo_completed = req.body.todo_completed;

            todo.save().then(todo => {
                res.json('Todo updated!');
            })
            .catch(err => {
                res.status(400).send("Update not possible");
            });
    });
});
```

This route is used to update an existing todo item (e.g. setting the todo_completed property to true). Again this path is containing a parameter: id. Inside the callback function which is passed into the call of post, we're first retrieving the old todo item from the database based on the id. Once the todo item is retrieved we're setting the todo property values to what's available in the request body. Finally we need to call todo.save to save the updated object in the database again.

Finally, in the following you can see the complete and final code of server.js again:

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const cors = require('cors');
const mongoose = require('mongoose');
const todoRoutes = express.Router();
const PORT = 4000;

let Todo = require('./todo.model');

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

mongoose.connect('mongodb://127.0.0.1:27017/todos', { useNewUrlParser: true });
const connection = mongoose.connection;

connection.once('open', function() {
    console.log("MongoDB database connection established successfully");
})

todoRoutes.route('/').get(function(req, res) {
    Todo.find(function(err, todos) {
        if (err) {
            console.log(err);
        } else {
            res.json(todos);
        }
    });
});

todoRoutes.route('/:id').get(function(req, res) {
    let id = req.params.id;
    Todo.findById(id, function(err, todo) {
        res.json(todo);
    });
});

todoRoutes.route('/update/:id').post(function(req, res) {
    Todo.findById(req.params.id, function(err, todo) {
        if (!todo)
            res.status(404).send("data is not found");
        else
            todo.todo_description = req.body.todo_description;
            todo.todo_responsible = req.body.todo_responsible;
            todo.todo_priority = req.body.todo_priority;
            todo.todo_completed = req.body.todo_completed;

            todo.save().then(todo => {
                res.json('Todo updated!');
            })
            .catch(err => {
                res.status(400).send("Update not possible");
            });
    });
});

todoRoutes.route('/add').post(function(req, res) {
    let todo = new Todo(req.body);
    todo.save()
        .then(todo => {
            res.status(200).json({'todo': 'todo added successfully'});
        })
        .catch(err => {
            res.status(400).send('adding new todo failed');
        });
});

app.use('/todos', todoRoutes);

app.listen(PORT, function() {
    console.log("Server is running on Port: " + PORT);
});
```

Testing The Server API With Postman

Having completed the server implementation we're now ready to run tests for our HTTP endpoints by using the Postman tool (https://www.getpostman.com/).

1) First let's add a first todo item to our database by sending a HTTP POST request

As a result we're getting returned a JSON object which is containing the message: todo added successfully. Having now added the first todo item to out database we're able to retrieve the list of todos which are available by sending an HTTP GET request to the /todos path:

Of course we're only getting back an array which is containing only one item (the todo item we've just inserted). From the response you can see that an id has been assigned automatically to this item.

Next let's use this id to test route /todos/:id to retrieve a single todo element based on it's id:

As expected the same todo item is returned in JSON format as before. Next, let's try to update the todo's todo_completed property by sending a POST request to the /update/:id path:

The body of this POST request has to include the todo item with all its properties (except _id and __v) in JSON format. Sending this request should return the message Todo updated!. Let's check if the value of todo_completed has been updated in the database by once again requesting the todo item by id:

As expected the todo item with the updated value is being returned in JSON format.

What's Next

In the first part of this series we've started to create the React front-end application for the MERN stack todo application. In this second part we've continued with building the back-end server based on Node.js, Express, and MongoDB. We've connected the Node.js / Express server to MongoDB by using the Mongoose library.

We've been using Postman to make sure that the server endpoints are working as expected. In the upcoming part we'll further complete the implementation of the front-end react application and also connect font- and back-end.

Part 3

This is the third part of The MERN Stack Tutorial – Building a React CRUD Application From Start To Finish series. In the second part we've finished the implementation of the back-end part by using Node.js, Express, and MongoDB. In this part we're now ready to return to the React front-end application (which we've started to implemented in the first part) and add the connection to the back-end, so that the user will be able to

create new todo items
see an overview of all todo items
The communication between front-end and back-end will be done by sending HTTP request to the various server endpoints we've created in the last part.

Installing Axios
In order to be able to send HTTP request to our back-end we're making use of the Axios library. Axios is being installed via the following command:

```
$ npm install axios
```

Once Axios is added to the project we're ready to further complete the implementation of CreateTodo component and send data to the back-end.

Completing The Implementation Of createTodo Component
First let's add the following import statement to create-todo.component.ts so that we're ready to use the Axios library in that file:

```
import axios from 'axios';
```
The right place where the code needs to be added which is responsible for sending the data of the new todo element to the back-end is the onSubmit method. The existing implementation of onSubmit needs to be extended in the following way:

```
  onSubmit(e) {
      e.preventDefault();

      console.log(`Form submitted:`);
      console.log(`Todo Description: ${this.state.todo_description}`);
      console.log(`Todo Responsible: ${this.state.todo_responsible}`);
      console.log(`Todo Priority: ${this.state.todo_priority}`);

      const newTodo = {
         todo_description: this.state.todo_description,
         todo_responsible: this.state.todo_responsible,
         todo_priority: this.state.todo_priority,
         todo_completed: this.state.todo_completed
      };
```

```
        axios.post('http://localhost:4000/todos/add', newTodo)
            .then(res => console.log(res.data));

        this.setState({
            todo_description: '',
            todo_responsible: '',
            todo_priority: '',
            todo_completed: false
        })
    }
```

Here we're using the axios.post method to send an HTTP POST request to the back-end endpoint http://localhost:4000/todos/add. This endpoint is expecting to get the new todo object in JSON format in the request body. Therefore we need to pass in the newTodo object as a second argument.

Let's fill out the create todo form:

Hit button Create Todo to send entered data to the server. By using Postman again, we're able to check if data has been stored in the MongoDB database. Let's again send a GET request to endpoint /todos and check if the new todo item is being returned:

Now that we've made sure that we're able to create new todo items from the React front-end application we're ready to move on and further complete the implementation of TodosList component in the next step.

Completing The Implementation Of TodosList Component

In todos-list.component.ts we start by adding the following import statement on top:

```
import { Link } from 'react-router-dom';
import axios from 'axios';
```

In the next step, let's use the component's constructor to initialize the state with an empty todos array:

```
    constructor(props) {
        super(props);
        this.state = {todos: []};
    }
```

To retrieve the todos data from the database the componentDidMount lifecycle method is added:

```
    componentDidMount() {
        axios.get('http://localhost:4000/todos/')
            .then(response => {
                this.setState({ todos: response.data });
            })
            .catch(function (error){
                console.log(error);
            })
    }
```

Here we're using the axios.get method to access the /todos endpoint. Once the result becomes available we're assigning response.data to the todos property of the component's state object by using the this.setState method.

Finally the JSX code needs to be added to the return statement of the render function like you can see in the following listing:

```
render() {
        return (
            <div>
                <h3>Todos List</h3>
                <table className="table table-striped" style={{ marginTop: 20 }} >
                    <thead>
                        <tr>
                            <th>Description</th>
                            <th>Responsible</th>
                            <th>Priority</th>
                            <th>Action</th>
                        </tr>
                    </thead>
                    <tbody>
                        { this.todoList() }
                    </tbody>
                </table>
            </div>
        )
    }
```

The output is done as a table and inside the tbody element we're making use of the todoList method to output a table row for each todo item. Because of that we need to add the implementation of todoList method to TodosList component as well:

```
    todoList() {
        return this.state.todos.map(function(currentTodo, i){
            return <Todo todo={currentTodo} key={i} />;
        })
    }
```

Inside this method we're iterating through the list of todo items by using the map function. Each todo item is output by using the Todo component which is not yet implemented. The current todo item is assigned to the todo property of this component.

To complete the code in todos-list.component.js we need to add the implementation of Todo component as well. In the following listing you can see the complete code:

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';
import axios from 'axios';

const Todo = props => (
    <tr>
        <td>{props.todo.todo_description}</td>
        <td>{props.todo.todo_responsible}</td>
        <td>{props.todo.todo_priority}</td>
        <td>
            <Link to={"/edit/"+props.todo._id}>Edit</Link>
        </td>
    </tr>
)

export default class TodosList extends Component {
    constructor(props) {
        super(props);
        this.state = {todos: []};
    }

    componentDidMount() {
        axios.get('http://localhost:4000/todos/')
            .then(response => {
                this.setState({ todos: response.data });
            })
            .catch(function (error){
                console.log(error);
            })
    }

    todoList() {
        return this.state.todos.map(function(currentTodo, i){
            return <Todo todo={currentTodo} key={i} />;
        })
    }

    render() {
        return (
            <div>
                <h3>Todos List</h3>
                <table className="table table-striped" style={{ marginTop: 20 }} >
                    <thead>
                        <tr>
                            <th>Description</th>
                            <th>Responsible</th>
                            <th>Priority</th>
                            <th>Action</th>
                        </tr>
                    </thead>
                    <tbody>
                        { this.todoList() }
                    </tbody>
                </table>
            </div>
        )
    }
}
```
Todo component is implemented as a functional React component. It outputs the table row which contains the values of the properties of the todo item passed into that component. Inside the Actions column of the table we're also outputting a link to /edit/:id route by using the Link component.

The resulting output should then look like the following:

What's Next
Now, we've connected the first parts of the React front-end application to the back-end. However, some part are still missing. In the upcoming and last part of this tutorial series we're going to further complete our front-end application so that the user will also be able to edit todo items and set todo items to completed.

Part 4

This is the fourth and last part of The MERN Stack Tutorial – Building a React CRUD Application From Start To Finish series.

In the last part we've connected the first pieces of the React front-end application to the back-end. However, some pieces are still missing. In this last part of this tutorial series we're going to further complete our front-end application so that the user will also be able to edit todo items and set todo items to completed. Let's get started …

Linking To EditTodo Component
The link to EditTodo component has already been included in the output which returned by Todo component (implemented in todos-list.component.js):

```
const Todo = props => (
    <tr>
        <td className={props.todo.todo_completed ? 'completed' : ''}>{props.todo.todo_description}
</td>
        <td className={props.todo.todo_completed ? 'completed' : ''}>{props.todo.todo_responsible}
</td>
        <td className={props.todo.todo_completed ? 'completed' : ''}>{props.todo.todo_priority}</td>
        <td>
            <Link to={"/edit/"+props.todo._id}>Edit</Link>
        </td>
    </tr>
)
```

The Edit link which is output for each todo entry is pointing to path /edit/:id. The ID of the current todo is included in the URL so that we're able to retrieve the current ID in the implementation of EditTodo component again.

Furthermore you can see that depending on the todo_completed value of a todo to completed CSS class is applied (if todo_completed is true). The CSS class is added in index.css:

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

Edit Todos
Ok, so let's turn to the implementation of EditTodo component in edit-todo.component.js. First of all the Axios library needs to be made available by adding the following import statement:

```
import axios from 'axios';
```

Next we're adding a class constructor to set the initial state:

```
    constructor(props) {
        super(props);

        this.state = {
            todo_description: '',
            todo_responsible: '',
            todo_priority: '',
            todo_completed: false
        }
    }
```

The state object is consisting of four properties which are representing one single todo item. To retrieve the current todo item (based on it’s ID) from the back-end and update the component’s state accordingly the componentDidMount lifecycle method is added in the following way:

```
    componentDidMount() {
        axios.get('http://localhost:4000/todos/'+this.props.match.params.id)
            .then(response => {
                this.setState({
                    todo_description: response.data.todo_description,
                    todo_responsible: response.data.todo_responsible,
                    todo_priority: response.data.todo_priority,
                    todo_completed: response.data.todo_completed
                })
            })
            .catch(function (error) {
                console.log(error);
            })
    }
```

Here we’re making use of Axios once again to send an HTTP GET request to the back-end in order to retrieve todo information. Because we’ve been handing over the ID as a URL parameter we’re able to access this information via this.props.match.params.id so that we’re able to pass on this information to the back-end.

The response which is returned from the back-end is the todo item the user has requested to edit. Once the result is available we’re setting the component’s state again with the values from the todo item received.

With the state containing the information of the todo item which has been selected to be edited we’re now ready to output the form, so that the user is able to see what’s available and is also able to use the form to alter data. As always the corresponding JSX code needs to be added to the return statement of the component’s render method.

```
 render() {
        return (
            <div>
                <h3 align="center">Update Todo</h3>
                <form onSubmit={this.onSubmit}>
                    <div className="form-group">
                        <label>Description: </label>
                        <input  type="text"
                                className="form-control"
                                value={this.state.todo_description}
                                onChange={this.onChangeTodoDescription}
                                />
                    </div>
                    <div className="form-group">
                        <label>Responsible: </label>
                        <input
                                type="text"
                                className="form-control"
                                value={this.state.todo_responsible}
                                onChange={this.onChangeTodoResponsible}
                                />
                    </div>
                    <div className="form-group">
            <div className="form-check form-check-inline">
                <input  className="form-check-input"
                        type="radio"
                        name="priorityOptions"
                        id="priorityLow"
                        value="Low"
                        checked={this.state.todo_priority==='Low'}
                        onChange={this.onChangeTodoPriority}
                        />
                <label className="form-check-label">Low</label>
            </div>
            <div className="form-check form-check-inline">
                <input  className="form-check-input"
                        type="radio"
                        name="priorityOptions"
                        id="priorityMedium"
                        value="Medium"
                        checked={this.state.todo_priority==='Medium'}
                        onChange={this.onChangeTodoPriority}
                        />
                <label className="form-check-label">Medium</label>
            </div>
            <div className="form-check form-check-inline">
                <input  className="form-check-input"
                        type="radio"
                        name="priorityOptions"
                        id="priorityHigh"
                        value="High"
                        checked={this.state.todo_priority==='High'}
                        onChange={this.onChangeTodoPriority}
                        />
                <label className="form-check-label">High</label>
            </div>
        </div>
        <div className="form-check">
            <input  className="form-check-input"
                    id="completedCheckbox"
                    type="checkbox"
                    name="completedCheckbox"
                    onChange={this.onChangeTodoCompleted}
                    checked={this.state.todo_completed}
                    value={this.state.todo_completed}
                    />
            <label className="form-check-label" htmlFor="completedCheckbox">
                Completed
            </label>
        </div>

        <br />

        <div className="form-group">
            <input type="submit" value="Update Todo" className="btn btn-primary" />
        </div>
    </form>
  </div>
)
```

```
    }
```
Herewith the following form is generated:
The form is using several event handler methods which are connected to the onChange event types of the input controls:

onChangeTodoDescription
onChangeTodoResponsible
onChangeTodoPriority
onChangeTodoCompleted
Furthermore the submit event of the form is bound to the onSubmit event handler method of the component.

The four onChange event handler methods are making sure that the state of the component is update everytime the user changes the input values of the form controls:

```
onChangeTodoDescription(e) {
        this.setState({
            todo_description: e.target.value
        });
    }

    onChangeTodoResponsible(e) {
        this.setState({
            todo_responsible: e.target.value
        });
    }

    onChangeTodoPriority(e) {
        this.setState({
            todo_priority: e.target.value
        });
    }

    onChangeTodoCompleted(e) {
        this.setState({
            todo_completed: !this.state.todo_completed
        });
    }
```
The onSubmit event handler method is creating a new todo object based on the values available in the component's state and then initiating a post request to the back-end endpoint http://localhost:4000/todos/update/:id to create a new todo item in the MongoDB database:

```
    onSubmit(e) {
        e.preventDefault();
        const obj = {
            todo_description: this.state.todo_description,
            todo_responsible: this.state.todo_responsible,
            todo_priority: this.state.todo_priority,
            todo_completed: this.state.todo_completed
        };
        console.log(obj);
        axios.post('http://localhost:4000/todos/update/'+this.props.match.params.id, obj)
            .then(res => console.log(res.data));

        this.props.history.push('/');
    }
```

By calling this.props.history.push('/') it is also made sure that the user is redirected back to the default route of the application, so that the list of todos is shown again.

Because we're accessing the component's state (this.state) in the event handler method we need to create a lexcial binding to this for all five methods in the constructor:

```
  constructor(props) {
      super(props);

      this.onChangeTodoDescription = this.onChangeTodoDescription.bind(this);
      this.onChangeTodoResponsible = this.onChangeTodoResponsible.bind(this);
      this.onChangeTodoPriority = this.onChangeTodoPriority.bind(this);
      this.onChangeTodoCompleted = this.onChangeTodoCompleted.bind(this);
      this.onSubmit = this.onSubmit.bind(this);

      this.state = {
          todo_description: '',
          todo_responsible: '',
          todo_priority: '',
          todo_completed: false
      }
  }
```

With these code changes in place we now should have a fully working MERN-stack application which allows us to:

View a list of todo items
Create new todo items
Update existing todo items
Set todo items to status completed
Finally, take a look at the following listing. Here you can see the complete and final code of edit-todo.component.js again:

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

export default class EditTodo extends Component {

    constructor(props) {
        super(props);

        this.onChangeTodoDescription = this.onChangeTodoDescription.bind(this);
        this.onChangeTodoResponsible = this.onChangeTodoResponsible.bind(this);
        this.onChangeTodoPriority = this.onChangeTodoPriority.bind(this);
        this.onChangeTodoCompleted = this.onChangeTodoCompleted.bind(this);
        this.onSubmit = this.onSubmit.bind(this);

        this.state = {
            todo_description: '',
            todo_responsible: '',
            todo_priority: '',
            todo_completed: false
        }
    }

    componentDidMount() {
        axios.get('http://localhost:4000/todos/'+this.props.match.params.id)
```

```
            .then(response => {
                this.setState({
                    todo_description: response.data.todo_description,
                    todo_responsible: response.data.todo_responsible,
                    todo_priority: response.data.todo_priority,
                    todo_completed: response.data.todo_completed
                })
            })
            .catch(function (error) {
                console.log(error);
            })
    }

    onChangeTodoDescription(e) {
        this.setState({
            todo_description: e.target.value
        });
    }

    onChangeTodoResponsible(e) {
        this.setState({
            todo_responsible: e.target.value
        });
    }

    onChangeTodoPriority(e) {
        this.setState({
            todo_priority: e.target.value
        });
    }

    onChangeTodoCompleted(e) {
        this.setState({
            todo_completed: !this.state.todo_completed
        });
    }

    onSubmit(e) {
        e.preventDefault();
        const obj = {
            todo_description: this.state.todo_description,
            todo_responsible: this.state.todo_responsible,
            todo_priority: this.state.todo_priority,
            todo_completed: this.state.todo_completed
        };
        console.log(obj);
        axios.post('http://localhost:4000/todos/update/'+this.props.match.params.id, obj)
            .then(res => console.log(res.data));

        this.props.history.push('/');
    }

    render() {
        return (
            <div>
                <h3 align="center">Update Todo</h3>
```

```
<form onSubmit={this.onSubmit}>
    <div className="form-group">
        <label>Description: </label>
        <input  type="text"
                className="form-control"
                value={this.state.todo_description}
                onChange={this.onChangeTodoDescription}
                />
    </div>
    <div className="form-group">
        <label>Responsible: </label>
        <input
                type="text"
                className="form-control"
                value={this.state.todo_responsible}
                onChange={this.onChangeTodoResponsible}
                />
    </div>
    <div className="form-group">
        <div className="form-check form-check-inline">
            <input  className="form-check-input"
                    type="radio"
                    name="priorityOptions"
                    id="priorityLow"
                    value="Low"
                    checked={this.state.todo_priority==='Low'}
                    onChange={this.onChangeTodoPriority}
                    />
            <label className="form-check-label">Low</label>
        </div>
        <div className="form-check form-check-inline">
            <input  className="form-check-input"
                    type="radio"
                    name="priorityOptions"
                    id="priorityMedium"
                    value="Medium"
                    checked={this.state.todo_priority==='Medium'}
                    onChange={this.onChangeTodoPriority}
                    />
            <label className="form-check-label">Medium</label>
        </div>
        <div className="form-check form-check-inline">
            <input  className="form-check-input"
                    type="radio"
                    name="priorityOptions"
                    id="priorityHigh"
                    value="High"
                    checked={this.state.todo_priority==='High'}
                    onChange={this.onChangeTodoPriority}
                    />
            <label className="form-check-label">High</label>
        </div>
    </div>
    <div className="form-check">
        <input  className="form-check-input"
                id="completedCheckbox"
                type="checkbox"
                name="completedCheckbox"
                onChange={this.onChangeTodoCompleted}
                checked={this.state.todo_completed}
                value={this.state.todo_completed}
                />
            <label className="form-check-label" htmlFor="completedCheckbox">
              Completed
            </label>
          </div>

          <br />

          <div className="form-group">
            <input type="submit" value="Update Todo" className="btn btn-primary" />
          </div>
        </form>
      </div>
    )
  }
}
```

Conclusion
The MERN stack combines MongoDB, Express, React and Node.js for back- and front-end web development. This four-part tutorial series provided you with a practical introduction to building a MERN stack application from start to finish. By building a simple todo manager application you've learnt how the various building blocks of the MERN stack are fitting together and are applied in a practical real-world application.