

CRUD con Persistencia

Ing. Luis Guillermo Molero Suárez

Parte 1: Configuración del proyecto

La pila MERN consta de las siguientes tecnologías:

- Node.js: Node.js es un tiempo de ejecución de JavaScript creado en el motor de JavaScript V8 de Chrome. Node.js trae JavaScript al servidor
- MongoDB: una base de datos de código abierto basada en documentos
- Express: un marco web minimalista, rápido y sin supervisión para Node.js
- React: una biblioteca de front-end de JavaScript para crear interfaces de usuario

La pila MERN es muy similar a la pila MEAN popular. La única diferencia aquí es que la pila MEAN usa Angular para construir la aplicación web front-end y la pila MERN usa React en su lugar.

La aplicación que se creará es una sencilla, será una aplicación de tareas pendientes. Al usar este ejemplo, es posible demostrar cómo construir una aplicación CRUD (Crear, Leer, Actualizar y Eliminar) desde cero usando la pila MERN.

En esta primera parte, se va a completar la configuración del proyecto React para construir el Front de la aplicación MERN. En la siguiente parte, se continuará con la implementación del servidor Node.js / Express.



Configuración de la aplicación React

Para este ejercicio, se asume que Node.js está instalado en su sistema. Si ese no es el caso, vaya primero a <https://nodejs.org/> y siga las instrucciones de instalación para su plataforma.

Puede verificar si Node.js está instalado en su sistema, escribiendo en su CMD lo siguiente:

```
node -v
```

En el siguiente paso, estamos creando el proyecto inicial de React usando el script create-react-app. Lo bueno de create-react-app es que este script se puede ejecutar usando el comando npx sin la necesidad de instalarlo primero en su sistema. Simplemente ejecute el siguiente comando:

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

La ejecución de este comando crea un nuevo directorio de proyecto mern-todo-app. Dentro de esta carpeta, encontrará la plantilla de proyecto React predeterminada con todas las dependencias instaladas.

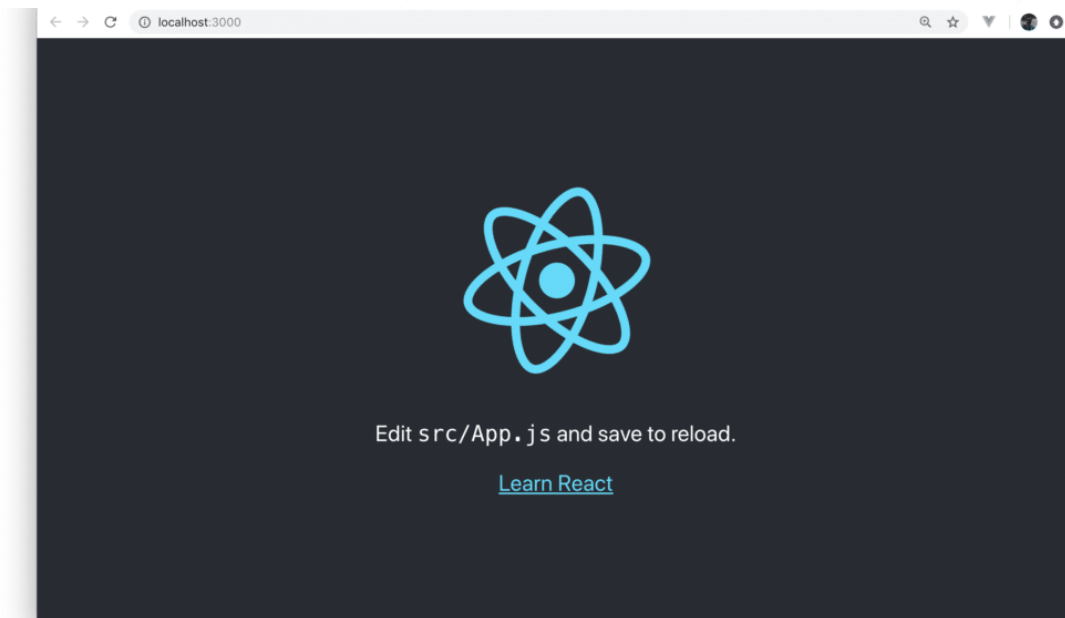
Cambie a la carpeta recién creada:

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

e inicie el servidor web de desarrollo ejecutando el siguiente comando:

```
npm start
```

Ahora debería poder ver el siguiente resultado en el navegador:



Agregar Bootstrap al proyecto React

A continuación, debemos agregar el framework Bootstrap a nuestro proyecto. Esto es necesario porque usaremos las clases CSS de Bootstrap para construir nuestra interfaz de usuario. Dentro de la carpeta del proyecto, ejecute el siguiente comando para agregar la biblioteca:

```
npm install bootstrap
```

A continuación, debe asegurarse de que el archivo CSS de Bootstrap se importe en App.js agregando la siguiente línea de código:

```
//src/App.js  
import "bootstrap/dist/css/bootstrap.min.css";
```

Además, debe deshacerse de la mayor parte del código predeterminado que está contenido en App.js, de modo que solo quede el siguiente código:

```
//src/App.js
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;
```

Configuración de React Router

Lo siguiente que debemos agregar al proyecto es el paquete React Router: react-router-dom:

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

Con este paquete instalado, estamos listos para agregar la configuración de enrutamiento en App.js.

En primer lugar, debe agregarse la siguiente declaración de importación:

```
//src/App.js
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
```

A continuación, insertemos el código JSX en un elemento `<Router>` `</Router>`:

```
//src/App.js
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;
```

Dentro del elemento `<Router>`, ahora estamos listos para agregar la configuración del enrutador dentro de ese elemento:

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

Para cada ruta que deba agregarse a la aplicación, se agrega un nuevo elemento `<Route>`. La ruta de atributos y el componente se utilizan para agregar los ajustes de configuración para cada ruta. Al usar la ruta de atributo, se establece la ruta de enrutamiento y al usar el atributo de componente, la ruta se conecta con un componente.

Como puede ver, necesitamos agregar tres rutas a nuestra aplicación:

- /
- /crear
- / editar /: id



Para estas tres rutas queremos conectarnos a tres componentes:

- TodosList
- EditTodo
- CreateTodo

Crear componentes

Para crear los componentes necesarios en nuestra aplicación, primero creemos un nuevo directorio `src/components` y creemos tres nuevos archivos:

`todos-list.component.js`

`edit-todo.component.js`

`create-todo.component.js`

Agreguemos una implementación básica del componente React para cada uno de esos componentes:

Primero, `todos-list-componen.js`

```
//src/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>
    )
  }
}
```

Segundo, edit-todo.component.js:

```
//src/components/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>
    )
  }
}
```

Tercero, create-todo.component.js:

```
//src/components/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>
    )
  }
}
```

Crear el diseño y la navegación básicos

A continuación, agreguemos un diseño básico y un menú de navegación a nuestra aplicación. Debido a que hemos agregado la biblioteca Bootstrap antes, ahora podemos hacer uso de las clases CSS de Bootstrap para implementar la interfaz de usuario de nuestra aplicación web.

Primero, copie “logo.png” en la carpeta “src” del proyecto. Este archivo está disponible en la siguiente ruta:

\\MisionTIC2022\\Modulo4_Web_MisionTIC2022_Main\\Semana_4\\Ejercicios\\images



Segundo, pegue el siguiente código en App.js

```
// src/App.js
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://github.com/luisguillermomolero/MisionTIC2022"
              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="collapse 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>
    );
  }
}
```



```
    </div>
  </Router>
);
}
}
export default App;
```

Revisemos nuevamente lo que podemos ver en el navegador

La barra de navegación se muestra con dos elementos de menú incluidos (Todos y Crear Todo). De forma predeterminada, se muestra la salida del componente TodosList porque estaba conectado a la ruta predeterminada de la aplicación.

Al hacer clic en el enlace Create Todo, se muestra la salida del componente CreateTodo

Implementación del componente create-todo-component.js

Agreguemos la implementación del componente createTodo en el archivo create-todo.component.js.

Primero comenzamos agregando un constructor a la clase del componente:

```
//src/components/create-todo.component.js:
constructor(props) {
  super(props);

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

Dentro del constructor, establecemos el estado inicial del componente asignando un objeto a this.state. El estado comprende las siguientes propiedades:

- todo_description
- todo_responsable
- todo_priority

- todo_completed

Además, necesitamos agregar métodos que se puedan usar para actualizar las propiedades del estado:

```
//src/components/create-todo.component.js:
onChangeTodoDescription(e) {
  this.setState({
    todo_description: e.target.value
  });
}

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

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

Finalmente, se necesita otro método para manejar el evento de envío del formulario que se implementará para crear un nuevo elemento de tareas pendientes:

```
//src/components/create-todo.component.js:
onSubmit(e) {
  e.preventDefault();

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

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

Dentro de este método, debemos llamar a `e.preventDefault` para asegurarnos de que se evite el comportamiento de envío del formulario HTML predeterminado. Debido a que el back-end de

nuestra aplicación aún no está implementado, solo imprimimos lo que está disponible actualmente en el estado del componente local en la consola. Finalmente, nos aseguramos de que el formulario se restablezca configurando el objeto de restablecimiento del estado.

Debido a que en los cuatro métodos implementados estamos tratando con el objeto de estado del componente, debemos asegurarnos de vincular esos métodos a este agregando las siguientes líneas de código al constructor:

```
//src/components/create-todo.component.js:
this.onChangeTodoDescription = this.onChangeTodoDescription.bind(this);
this.onChangeTodoResponsible = this.onChangeTodoResponsible.bind(this);
this.onChangeTodoPriority = this.onChangeTodoPriority.bind(this);
this.onSubmit = this.onSubmit.bind(this);
```

Finalmente, necesitamos agregar el código JSX que se necesita para mostrar el formulario:

```
//src/components/create-todo.component.js:
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>
)
}
```

A continuación, puede ver el código fuente completo que ahora debería estar disponible en `create-todo.component.js`:

```
//src/components/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>
            </form>
        </div>
    );
}
```

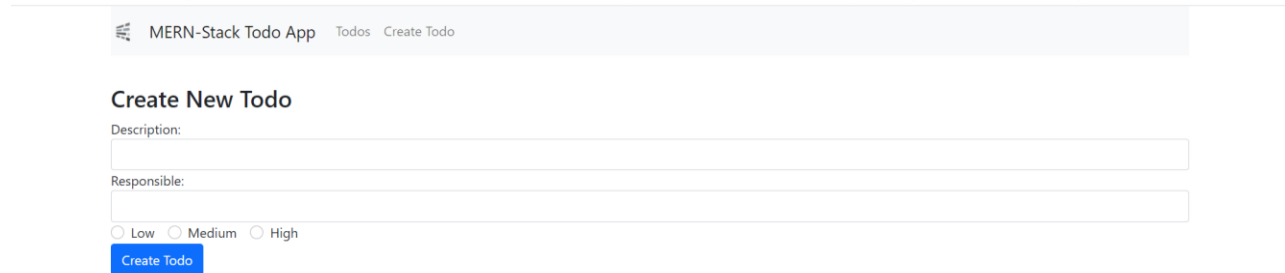


```
</div>
<div className="form-group">
  <label>Responsible: </label>
  <input
    type="text"
    className="form-control"
    value={this.state.todo_responsible}
    onChange={this.onChangeTodoResponsable}
  />
</div>
<div className="form-group">
  <div className="form-check form-check-inline">
    <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>
)
```

```
}  
}
```

El resultado que se entrega en el navegador al acceder a la ruta /create debe ser lo siguiente:



MERN-Stack Todo App Todos Create Todo

Create New Todo

Description:

Responsible:

☐ Low ☐ Medium ☐ High

Al completar el formulario y enviarlo haciendo clic en el botón “Create Todo”, se mostrarán los datos que se ingresaron en la consola.

Parte 2: Configurar el back-end

En la primera parte, comenzamos a implementar la aplicación React front-end de la aplicación MERN Todo. En esta segunda parte, nos centraremos en el back-end y crearemos un servidor utilizando Node.js / Express.

Al construir el back-end, también configuraremos MongoDB y nos conectaremos a la base de datos desde nuestro servidor Node.js / Express mediante el uso de la biblioteca Mongoose.

El back-end comprenderá puntos finales HTTP para cubrir los siguientes casos de uso:

- Recupere la lista completa de elementos pendientes disponibles enviando una solicitud HTTP GET
- Recupere un elemento de tarea específico mediante el envío de una solicitud HTTP GET y proporcione el ID de tarea específico, además
- Cree un nuevo elemento de tareas pendientes en la base de datos enviando una solicitud HTTP POST
- Actualice un elemento pendiente existente en la base de datos enviando una solicitud HTTP POST

Inicio del proyecto back-end

Para iniciar el proyecto de back-end, creemos una nueva carpeta de proyecto vacía:

```
mkdir backend
```

Cambie a esa carpeta recién creada usando:

```
cd backend
```

Creemos un archivo package.json dentro de esa carpeta con el siguiente comando:

```
npm init -y
```

Con el archivo package.json disponible en la carpeta del proyecto, estamos listos para agregar algunas dependencias al proyecto:

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

Echemos un vistazo rápido a los cuatro paquetes:

- **express:** Express es un marco web rápido y ligero para Node.js. Express es una parte esencial de la pila MERN.
- **body-parser:** middleware de análisis corporal de Node.js.
- **cors:** CORS es un paquete node.js para proporcionar un middleware Express que se puede usar para habilitar CORS con varias opciones. El intercambio de recursos de origen cruzado (CORS) es un mecanismo que permite solicitar recursos restringidos en una página web desde otro dominio fuera del dominio desde el que se sirvió el primer recurso.
- **mongoose:** Un marco Node.js que nos permite acceder a MongoDB de una manera orientada a objetos.

Finalmente, debemos asegurarnos de instalar un paquete global ejecutando el siguiente comando:

```
npm install -g nodemon
```

Nodemon es una utilidad que monitoreará cualquier cambio en su fuente y reiniciará automáticamente su servidor. Usaremos nodemon cuando ejecutemos nuestro servidor Node.js en los siguientes pasos.

Dentro de la carpeta del proyecto de backend, cree un nuevo archivo llamado server.js e inserte la siguiente implementación básica del servidor Node.js / Express:


```
//backend/server.js
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);
});
```

Con este código, estamos creando un servidor Express, adjuntando el middleware cors y body-parser y haciendo que el servidor escuche en el puerto 4000.

Inicie el servidor utilizando nodemon:

```
nodemon server
```

Ahora debería ver un resultado similar al siguiente:

```
C:\Windows\System32\cmd.exe - nodemon server
added 81 packages from 126 contributors and audited 81 packages in 5.621s

2 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

D:\PRACTICAS\SEMANA_4\mern-todo-app\backend>npm install -g nodemon
C:\Users\luisg\AppData\Roaming\npm\nodemon -> C:\Users\luisg\AppData\Roaming\npm\node_modules\nodemon\bin\nodemon.js

> nodemon@2.0.12 postinstall C:\Users\luisg\AppData\Roaming\npm\node_modules\nodemon
> node bin/postinstall || exit 0

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@~2.3.2 (node_modules\nodemon\node_modules\chokidar\node_modules
\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ nodemon@2.0.12
added 119 packages from 53 contributors in 4.863s

D:\PRACTICAS\SEMANA_4\mern-todo-app\backend>nodemon server
[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server is running on Port: 4000
```



Como podemos ver que el servidor de salida se está ejecutando en el puerto: 4000, sabemos que el servidor se ha iniciado correctamente y está listado en el puerto 4000.

Instalación de MondoDB

Ahora que logramos configurar un servidor Node.js/Express básico, estamos listos para continuar con la siguiente tarea: configurar la base de datos MongoDB.

En primer lugar, debemos asegurarnos de que MongoDB esté instalado en su sistema. Si está trabajando en Windows, siga las instrucciones de instalación de

<https://docs.mongodb.com/manual/administration/install-community/>

Una vez instalado MongoDB en su sistema, debe crear un directorio /data/db en la carpeta “backend” de datos que utilice MongoDB, de la forma:

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

Antes de ejecutar mongod por primera vez, asegúrese de que la cuenta de usuario que ejecuta mongod tenga permisos de lectura y escritura para el directorio.

Ahora estamos listos para iniciar MongoDB ejecutando el siguiente comando:

```
mongod
```

La ejecución de este comando le dará el siguiente resultado en la línea de comando:

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.19042.1083]
(c) Microsoft Corporation. Todos los derechos reservados.

D:\PRACTICAS\SEMANA_4\mern-todo-app\backend>mongod
{"t":{"$date":"2021-07-11T10:07:53.413-05:00"},"s":"I", "c":"CONTROL", "id":23285, "ctx":{"main","msg":"Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"}}
{"t":{"$date":"2021-07-11T10:07:53.434-05:00"},"s":"W", "c":"ASIO", "id":22601, "ctx":{"main","msg":"No TransportLayer configured during NetworkInterface startup"}}
{"t":{"$date":"2021-07-11T10:07:53.436-05:00"},"s":"I", "c":"NETWORK", "id":4000002, "ctx":{"main","msg":"Implicit CXX runtime in use: /usr/bin/cxx"}}
{"t":{"$date":"2021-07-11T10:07:53.439-05:00"},"s":"I", "c":"STORAGE", "id":4615611, "ctx":{"initandlisten","msg":"MongoDB starting", "attr":{"pid":19316,"port":27017,"dbPath":"D:/data/db/", "architecture":"64-bit", "host":"DESKTOP-4C8BAGB"}}}
{"t":{"$date":"2021-07-11T10:07:53.439-05:00"},"s":"I", "c":"CONTROL", "id":23403, "ctx":{"initandlisten","msg":"Build info", "attr":{"buildInfo":{"version":"4.4.6","gitVersion":"72e66213c2c3eab37d9358d5e78ad7f5c1d0b0d7","modules":[],"allocator":"tcmalloc","environment":{"distmod":"windows","distarch":"x86_64","target_arch":"x86_64"}}}}}
{"t":{"$date":"2021-07-11T10:07:53.439-05:00"},"s":"I", "c":"CONTROL", "id":51765, "ctx":{"initandlisten","msg":"Operating System", "attr":{"os":{"name":"Microsoft Windows 10","version":"10.0 (build 19042)"}}}}
{"t":{"$date":"2021-07-11T10:07:53.439-05:00"},"s":"I", "c":"CONTROL", "id":21951, "ctx":{"initandlisten","msg":"Options set by command line", "attr":{"options":{}}}}
{"t":{"$date":"2021-07-11T10:07:53.444-05:00"},"s":"E", "c":"STORAGE", "id":20557, "ctx":{"initandlisten","msg":"DBException in initAndlisten, terminating", "attr":{"error":{"NonExistentPath: Data directory D:\data\db\ not found. Create the missing directory or specify another path using (1) the --dbpath command line option, or (2) by adding the 'storage.dbPath' option in the configuration file."}}}}
{"t":{"$date":"2021-07-11T10:07:53.445-05:00"},"s":"I", "c":"REPL", "id":4784900, "ctx":{"initandlisten","msg":"Stepping down the ReplicationCoordinator for shutdown", "attr":{"waitTimeMillis":10000}}}
{"t":{"$date":"2021-07-11T10:07:53.449-05:00"},"s":"I", "c":"COMMAND", "id":4784901, "ctx":{"initandlisten","msg":"Shutting down the MirrorMaster"}}
{"t":{"$date":"2021-07-11T10:07:53.450-05:00"},"s":"I", "c":"SHARDING", "id":4784902, "ctx":{"initandlisten","msg":"Shutting down the WaitForMajorityService"}}
{"t":{"$date":"2021-07-11T10:07:53.451-05:00"},"s":"I", "c":"NETWORK", "id":20562, "ctx":{"initandlisten","msg":"Shutdown: going to close listening sockets"}}
{"t":{"$date":"2021-07-11T10:07:53.451-05:00"},"s":"I", "c":"NETWORK", "id":4784905, "ctx":{"initandlisten","msg":"Shutting down the global connection pool"}}
{"t":{"$date":"2021-07-11T10:07:53.451-05:00"},"s":"I", "c":"STORAGE", "id":4784906, "ctx":{"initandlisten","msg":"Shutting down the FlowControlLicketholder"}}
{"t":{"$date":"2021-07-11T10:07:53.452-05:00"},"s":"I", "c":"-", "id":20520, "ctx":{"initandlisten","msg":"Stopping further FlowControl ticket acquisitions."}}
{"t":{"$date":"2021-07-11T10:07:53.452-05:00"},"s":"I", "c":"NETWORK", "id":4784918, "ctx":{"initandlisten","msg":"Shutting down the ReplicaSetMonitor"}}
{"t":{"$date":"2021-07-11T10:07:53.452-05:00"},"s":"I", "c":"SHARDING", "id":4784921, "ctx":{"initandlisten","msg":"Shutting down the MigrationUtilExecutor"}}
{"t":{"$date":"2021-07-11T10:07:53.454-05:00"},"s":"I", "c":"CONTROL", "id":4784925, "ctx":{"initandlisten","msg":"Shutting down free monitoring"}}
{"t":{"$date":"2021-07-11T10:07:53.454-05:00"},"s":"I", "c":"STORAGE", "id":4784927, "ctx":{"initandlisten","msg":"Shutting down the HealthLog"}}
{"t":{"$date":"2021-07-11T10:07:53.454-05:00"},"s":"I", "c":"STORAGE", "id":4784929, "ctx":{"initandlisten","msg":"Acquiring the global lock for shutdown"}}
{"t":{"$date":"2021-07-11T10:07:53.455-05:00"},"s":"I", "c":"-", "id":4784931, "ctx":{"initandlisten","msg":"Dropping the scope cache for shutdown"}}
{"t":{"$date":"2021-07-11T10:07:53.455-05:00"},"s":"I", "c":"FTDC", "id":4784926, "ctx":{"initandlisten","msg":"Shutting down full-time data capture"}}
{"t":{"$date":"2021-07-11T10:07:53.455-05:00"},"s":"I", "c":"CONTROL", "id":20965, "ctx":{"initandlisten","msg":"Now exiting"}}
{"t":{"$date":"2021-07-11T10:07:53.456-05:00"},"s":"I", "c":"CONTROL", "id":23138, "ctx":{"initandlisten","msg":"Shutting down", "attr":{"exitCode":100}}}
```

Esto muestra que la base de datos se está ejecutando ahora en el puerto 27017 y está esperando aceptar conexiones de cliente.

Creación de una nueva base de datos MongoDB

El siguiente paso es crear la instancia de la base de datos MongoDB.

Así, nos vamos a la carpeta “db” a fin de conectarnos al servidor de la base de datos usando el cliente MondoDB en la línea de comando, de la forma:

```
mongo
```

Una vez que se inicia el cliente, le solicita que ingrese los comandos de la base de datos. Al usar el siguiente comando, estamos creando una nueva base de datos con el nombre todos:

```
use todos
```

Conexión a MongoDB mediante Mongoose

Regresemos a la implementación del servidor Node.js/Express en server.js. Con el servidor de base de datos MongoDB en ejecución, ahora estamos listos para conectarnos a MongoDB desde nuestro programa de servidor mediante el uso de la biblioteca Mongoose.

Cambie la implementación en server.js a lo siguiente:

```
//backend/server.js
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);  
});
```

En la consola, ahora también debería ver la conexión de salida de la base de datos MongoDB establecida con éxito.

```
C:\Windows\System32\cmd.exe - nodemon server  
  
+ nodemon@2.0.12  
added 119 packages from 53 contributors in 4.863s  
  
D:\PRACTICAS\SEMANA_4\mern-todo-app\backend>nodemon server  
[nodemon] 2.0.12  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node server.js`  
Server is running on Port: 4000  
[nodemon] restarting due to changes...  
[nodemon] restarting due to changes...  
[nodemon] starting `node server.js`  
[nodemon] clean exit - waiting for changes before restart  
[nodemon] restarting due to changes...  
[nodemon] starting `node server.js`  
[nodemon] restarting due to changes...  
[nodemon] restarting due to changes...  
[nodemon] starting `node server.js`  
(node:14468) [MONGODB DRIVER] Warning: Current Server Discovery and Monitoring engine is deprecated, and will be removed  
in a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to  
the MongoClient constructor.  
(Use `node --trace-warnings ...` to show where the warning was created)  
Server is running on Port: 4000  
MongoDB database connection established successfully
```

Crear un Mongoose Schema

Al usar Mongoose, podemos acceder a la base de datos de MongoDB de una manera orientada a objetos. Esto significa que necesitamos agregar un esquema de Mongoose para nuestra entidad Todo a la implementación de nuestro proyecto a continuación.

Dentro de la carpeta del proyecto backend, cree un nuevo archivo `todo.model.js` e inserte las siguientes líneas de código para crear un esquema `Todo`:

```
//backend/todo.model.js
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);
```

Con este código en su lugar, ahora estamos listos para acceder a la base de datos MongoDB usando el esquema `Todo`.

Implementación de los puntos finales del servidor

En el último paso, completemos la implementación del servidor en `server.js` utilizando el esquema `Todo` que acabamos de agregar para implementar los puntos finales de API que nos gustaría proporcionar.

Para configurar los puntos finales, necesitamos crear una instancia del Express Router agregando la siguiente línea de código:

```
//backend/server.js
const todoRoutes = express.Router();
```

El enrutador se agregará como middleware y tomará el control de la solicitud comenzando con path/todos:

```
//backend/server.js
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);
    }
  });
});
```

La función que se pasa a la llamada del método get se usa para manejar la solicitud HTTP GET entrante en la ruta URL/todos/. En este caso, llamamos a Todos.find para recuperar una lista de todos los elementos de tareas pendientes de la base de datos de MongoDB. Nuevamente, la llamada de los métodos de búsqueda toma un argumento: una función de devolución de llamada que se ejecuta una vez que el resultado está disponible. Aquí nos aseguramos de que los resultados (disponibles en todos) se agreguen en formato JSON al cuerpo de la respuesta llamando a res.json(todos).

El siguiente punto final que debe implementarse es /:id. Esta extensión de ruta se utiliza para recuperar un elemento de tarea al proporcionar un ID. La lógica de implementación es sencilla:

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

Aquí aceptamos el ID del parámetro de URL al que se puede acceder a través de req.params.id. Esta identificación se pasa a la llamada de Tood.findById para recuperar un elemento de problema en función de su identificación. Una vez que el objeto todo está disponible, se adjunta a la respuesta HTTP en formato JSON.

A continuación, agreguemos la ruta que se necesita para poder agregar nuevos elementos de tareas pendientes enviando una solicitud de publicación HTTP (/ agregar):

```
//backend/server.js
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');
    });
});
```

El nuevo elemento de todo es parte del cuerpo de la solicitud HTTP POST, por lo que podemos acceder a la vista req.body y, con ello, crear una nueva instancia de Todo. Este nuevo elemento se guarda en la base de datos llamando al método save.

Finalmente, se agrega una ruta /actualización/: id HTTP POST:

```
//backend/server.js
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");
      });
  });
});
```

Esta ruta se utiliza para actualizar un elemento de todo existente (por ejemplo, establecer la propiedad todo_completed en verdadero). Nuevamente, esta ruta contiene un parámetro: id. Dentro de la función de devolución de llamada que se pasa a la llamada de publicación, primero recuperamos el elemento de tarea anterior de la base de datos en función de la identificación. Una

vez que se recupera el elemento de tarea pendiente, establecemos los valores de la propiedad de tarea pendiente según lo que esté disponible en el cuerpo de la solicitud. Finalmente necesitamos llamar a `todo.save` para guardar el objeto actualizado en la base de datos nuevamente.

Finalmente, a continuación, puede ver el código completo y final de `server.js` nuevamente:

```
//backend/server.js
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);
});
```

Parte 3: Conexión de front-end a back-end

En esta parte, ahora estamos listos para volver a la aplicación de front-end React y agregar la conexión al back-end, para que el usuario pueda:

- crear nuevos elementos de tareas pendientes.
- ver una descripción general de todos los elementos de tareas pendientes.

La comunicación entre el front-end y el back-end se realizará enviando una solicitud HTTP a los distintos puntos finales del servidor que hemos creado en la última parte.

Instalación de Axios

Para poder enviar solicitudes HTTP a nuestro backend, usamos la biblioteca Axios.

Axios se instalará en nuestra carpeta del proyecto /mern-todo-app mediante el siguiente comando:

```
npm install axios
```

Una vez que se agrega Axios al proyecto, estamos listos para completar la implementación del componente CreateTodo y enviar datos al back-end.

Completando la implementación del componente createTodo

Primero agreguemos la siguiente declaración de importación a create-todo.component.js para que estemos listos para usar la biblioteca Axios en ese archivo:

```
//components/create-todo.component.js  
import axios from 'axios';
```

El lugar correcto donde se debe agregar el código que es responsable de enviar los datos del nuevo elemento todo al back-end es el método onSubmit.

La implementación existente de onSubmit debe ampliarse de la siguiente manera:

```
//components/create-todo.component.js  
onSubmit(e) {  
  e.preventDefault();  
  
  console.log(`Form submitted`);  
  console.log(`Todo Description: ${this.state.todo_description}`);  
  console.log(`Todo Responsable: ${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  
  })  
}
```

Aquí estamos usando el método `axios.post` para enviar una solicitud HTTP POST al extremo de back-end `http://localhost:4000/todos/add`. Este punto final espera obtener el nuevo objeto todo en formato JSON en el cuerpo de la solicitud. Por lo tanto, debemos pasar el objeto `newTodo` como segundo argumento.

Completando la implementación del componente `TodosList`

En `todos-list.component.ts` comenzamos agregando la siguiente declaración de importación en la parte superior:

```
//components/todos-list.component.js
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: []};
  }
```

Para recuperar los datos de todos de la base de datos, se agrega el método de ciclo de vida `componentDidMount`:

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

Aquí estamos usando el método `axios.get` para acceder al punto final `/ todos`. Una vez que el resultado está disponible, asignamos `response.data` a la propiedad `todos` del objeto de estado del componente mediante el método `this.setState`.

Finalmente, el código JSX debe agregarse a la declaración de retorno de la función de renderizado, como puede ver en la siguiente lista:

```
//components/todos-list.component.js
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>
  )
}
```

La salida se realiza como una tabla y dentro del elemento tbody estamos haciendo uso del método todoList para generar una fila de tabla para cada elemento de todo. Por eso también necesitamos agregar la implementación del método todoList al componente TodosList:

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

Dentro de este método, estamos iterando a través de la lista de elementos de tareas pendientes utilizando la función de mapa. Cada elemento de todo se genera mediante el uso del componente Todo que aún no se ha implementado. El elemento de tarea actual se asigna a la propiedad de tarea de este componente.

Para completar el código en todos-list.component.js también necesitamos agregar la implementación del componente Todo. En el siguiente listado puede ver el código completo:

```
//components/todos-list.component.js
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>
            </tr>
          </thead>
          <tbody>
            {this.todoList()}
          </tbody>
        </table>
      </div>
    )
  }
}
```

```
        <th>Action</th>
      </tr>
    </thead>
    <tbody>
      { this.todoList() }
    </tbody>
  </table>
</div>
)
}
```

El componente Todo se implementa como un componente funcional de React. Genera la fila de la tabla que contiene los valores de las propiedades del elemento de tarea que se pasó a ese componente. Dentro de la columna Acciones de la tabla también estamos generando un enlace a /edit/: id route utilizando el componente Link.

Parte 4: Finalización de la aplicación

En esta última parte, vamos a completar aún más nuestra aplicación front-end para que el usuario también pueda editar elementos de tareas pendientes y configurar elementos de tareas pendientes como completados.

Vinculación al componente EditTodo

El enlace al componente EditTodo ya se ha incluido en la salida que devolvió el componente Todo (implementado en todos-list.component.js):

```
//components/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>
)
```

El enlace Editar que se genera para cada entrada de tareas pendientes apunta a la ruta / editar /: id.

El ID de la tarea actual se incluye en la URL para que podamos recuperar el ID actual en la implementación del componente EditTodo nuevamente.

Además, puede ver que, dependiendo del valor de todo_completed de un todo a completado, se aplica la clase CSS (si todo_completed es verdadero).

La clase CSS se agrega en index.css:

```
//src/index.css
.completed {
  text-decoration: line-through;
}
```

Al aplicar esta clase de CSS, la información de texto de un elemento de tarea se muestra con texto tachado

EditTodo

Bien, pasemos a la implementación del componente EditTodo en edit-todo.component.js. En primer lugar, la biblioteca Axios debe estar disponible agregando la siguiente declaración de importación:

```
//components/ edit-todo.component.js
import axios from 'axios';
```

A continuación, agregamos un constructor de clases para establecer el estado inicial:

```
//components/ edit-todo.component.js
constructor(props) {
  super(props);

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

El objeto de estado consta de cuatro propiedades que representan un solo elemento de tarea pendiente. Para recuperar el elemento de tarea actual (en función de su ID) del back-end y actualizar el estado del componente en consecuencia, el método de ciclo de vida `componentDidMount` se agrega de la siguiente manera:

```
//components/ edit-todo.component.js
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);
    })
}
```

Aquí estamos haciendo uso de Axios una vez más para enviar una solicitud HTTP GET al back-end con el fin de recuperar la información de todo. Debido a que hemos estado entregando el ID como un parámetro de URL, podemos acceder a esta información a través de `this.props.match.params.id` para que podamos pasar esta información al back-end.

La respuesta que se devuelve desde el back-end es el elemento de tareas pendientes que el usuario ha solicitado editar. Una vez que el resultado está disponible, volvemos a configurar el estado del componente con los valores del elemento de tarea recibido.

Con el estado que contiene la información del elemento de tarea que se ha seleccionado para ser editado, ahora estamos listos para generar el formulario, de modo que el usuario puede ver lo que está disponible y también puede usar el formulario para modificar los datos. Como siempre, el código JSX correspondiente debe agregarse a la declaración de retorno del método de representación del componente.



```
//components/ edit-todo.component.js
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"
            />
          </div>
        </div>
      </form>
    </div>
  );
}
```

```
        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>
)
```

A continuación, se genera el siguiente formulario. Este, utiliza varios métodos de manejo de eventos que están conectados a los tipos de eventos onChange de los controles de entrada:

- onChangeTodoDescription
- onChangeTodoResponsable
- onChangeTodoPriority
- onChangeTodoCompleted

Además, el evento de envío del formulario está vinculado al método del controlador de eventos onSubmit del componente.

Los cuatro métodos del controlador de eventos onChange se aseguran de que el estado del componente se actualice cada vez que el usuario cambia los valores de entrada de los controles de formulario:

```
//components/ edit-todo.component.js
onChangeTodoDescription(e) {
  this.setState({
    todo_description: e.target.value
  });
}

onChangeTodoResponsable(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
  });
}
```

El método del controlador de eventos onSubmit está creando un nuevo objeto todo basado en los valores disponibles en el estado del componente y luego iniciando una solicitud de publicación al endpoint de back-end `http://localhost:4000/todos/update/:id` para crear un nuevo todo elemento en la base de datos MongoDB:

```
//components/ edit-todo.component.js
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('/');
}
```

Al llamar a `this.props.history.push("/")` también se asegura de que el usuario sea redirigido a la ruta predeterminada de la aplicación, de modo que la lista de todos se muestre nuevamente. Debido a que estamos accediendo al estado del componente (`this.state`) en el método del controlador de eventos, necesitamos crear un enlace lexical a `this` para los cinco métodos en el constructor:

```
//components/ edit-todo.component.js
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
  }
}
```

Con estos cambios de código en su lugar, ahora deberíamos tener una aplicación de pila MERN completamente funcional que nos permita:

- Ver una lista de elementos de tareas pendientes
- Crear nuevos elementos de tareas pendientes
- Actualizar elementos pendientes existentes
- Establecer elementos de tareas pendientes en el estado completado

Finalmente, eche un vistazo a la siguiente lista. Aquí puede ver el código completo y final de `edit-todo.component.js` nuevamente:

```
//components/ edit-todo.component.js
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"
            />
          </div>
        </div>
      </form>
    </div>
  );
}
```



```
        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>
)
}
```

Ahora, abrir un CMD en la carpeta backend y ejecute el comando para iniciar MongoDB, le permitirá guardar los datos del estudiante en la base de datos:

```
mongod
```

Seguidamente, ejecutar el siguiente comando para iniciar el servidor Nodemon.

```
npx nodemon server.js
```

Finalmente, en la carpeta de proyecto ejecutar:

```
yarn start
```

Testing React con Jest y Enzyme

En el proyecto, abrir un CMD y teclear lo siguiente:

```
Npm install -save-dev enzyme react-addons-test-utils
```

Ahora, nos vamos al archivo App.test.js y observamos el test por defecto



Seguidamente, abrir un CMD y teclear lo siguiente:

```
npm test
```