Creación de una API REST

Desarrollo de Sistemas Web - Back End

Versión 1.0 F. Act 27/09/2024

Actividad

Ahora que conocemos el funcionamiento de Mongodb y Mongoose, podemos crear una API REST completa que persista datos como corresponde. Para el ejemplo, vamos a crear una API que gestione series de TV, por lo que debemos crear los modelos y la estructura necesaria.

Creación del proyecto y estructura

En la terminal escribimos: **npm init**

Después, con el siguiente comando procederemos a instalar express, mongoose, body-parser (no obligatorio en versiones nuevas) y method-override

npm install --save express mongoose body-parser method-override

Se agregan las dependencias instaladas al archivo package.json

Estas son las librerías instaladas:

- express: La librería para Node.js que ya hemos visto.
- mongoose: La librería para MongoDB para interactuar con la base de datos, como ya hemos visto.
- body-parser: Un middleware para express que analiza el cuerpo de las solicitudes HTTP y lo convierte en un objeto JSON accesible. Permite recibir y procesar datos enviados en el cuerpo de una solicitud HTTP.
- method-override: Un middleware para express que permite sobrescribir el método HTTP utilizado en una solicitud.

Body-parser

Es un middleware de Node.js utilizado en aplicaciones Express para facilitar la lectura y el procesamiento de los cuerpos de las solicitudes HTTP. Permite que tu aplicación interprete los datos que llegan en el cuerpo de las solicitudes, especialmente en formatos como JSON, URL-encoded, entre otros.

Tipos de análisis:

JSON: bodyParser.json() analiza el cuerpo de la solicitud cuando se envía en formato JSON.

URL-encoded: bodyParser.urlencoded({ extended: true }) analiza los datos que se envían a través de formularios HTML y los convierte en un objeto JavaScript.

¿Por qué usarlo? Facilita el acceso a los datos en el cuerpo de la solicitud. Reduce la cantidad de código necesario para manejar diferentes tipos de datos. Mejora la legibilidad y organización de tu código.

Notas: A partir de Express 4.16.0, no es necesario instalar body-parser por separado, ya que sus funciones se integraron directamente en Express. Puedes usar express.json() y express.urlencoded() para el mismo propósito.

Method-override

Es un middleware de Node.js que permite cambiar el método HTTP de las solicitudes. Esto es especialmente útil en situaciones donde los navegadores solo permiten métodos GET y POST. Con method-override, puedes simular métodos como PUT o DELETE, lo cual es común en aplicaciones RESTful.

¿Por qué usarlo?

- Permite usar métodos HTTP que de otro modo no estarían disponibles en un formulario HTML.
- Facilita la construcción de API RESTful.
- Mejora la semántica de las solicitudes HTTP, haciendo que sean más claras y fáciles de entender.

Notas:

- Puedes personalizar el nombre del parámetro utilizado para especificar el método, por defecto es _method, pero puedes cambiarlo al configurarlo.

package.json

```
{
    "name": "api-rest-series",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {
        "body-parser": "^1.20.1",
        "express": "^4.18.2",
        "method-override": "^3.0.0",
        "mongoose": "^6.8.3"
    }
}
```

Inicializando el servidor HTTP

Las primeras líneas se encargan de incluir las dependencias que vamos a usar con require. Importamos Express para facilitarnos crear el servidor y realizar llamadas HTTP, Creamos un archivo llamado app.js en el directorio raíz que será el que ejecute nuestra aplicación y arranque nuestro servidor. Crearemos en primer lugar un sencillo servidor web para comprobar que tenemos todo lo necesario instalado, y a continuación iremos escribiendo más código.

Por otro lado les dejo un listado de código http de los más usados para que los puedan colocar para el manejo de errores.
https://es.semrush.com/blog/codigos-de-estado-http/

```
const express = require('express');
const app = express();

app.get('/', (request, response) => {
  response.send('
');
});

const PORT = 8000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Agregar Nodemon

Como hemos visto, Nodemon es una herramienta de desarrollo que automáticamente reinicia el servidor cada vez que se hacen cambios en el código. Esto es útil durante el desarrollo, ya que permite ver los cambios en el código sin tener que detener y volver a iniciar manualmente el servidor cada vez.

En la terminal: **npm install nodemon --save-dev** y ejecutamos con **npm run dev**

```
{
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "nodemon app.js"
  }
}
```

Creando los modelos de nuestra API REST

```
// Define el esquema para la colección de series de TV.
const tvShowSchema = new mongoose.Schema({
  title: { type: String, required: true },
  year: { type: Number, required: true },
  country: { type: String, required: true },
  poster: { type: String, required: true },
  seasons: { type: Number, required: true },
  genre: {
    type: String,
    enum: ['Drama', 'Fantasy', 'Sci-Fi', 'Thriller', 'Comedy']
  },
  summary: { type: String }
});

// Registra el modelo 'TVShow' basado en el esquema definido.
const TVShow = mongoose.model('TVShow', tvShowSchema);

// Exporta el modelo para que pueda ser utilizado en la aplicación.
module.exports = TVShow;
```

Vamos a crear un modelo usando Mongoose para poder guardar la información en la base de datos siguiendo el esquema.

Como dijimos, para este ejemplo vamos a crear una base de datos de series de TV, por lo tanto vamos a crear un modelo (en el archivo: models/tvshow.js) que incluya la información de una serie de TV, como pueden ser su título, el año de inicio, país de producción, una imagen promocional, número de temporadas, género y resumen del argumento.

En el ejemplo figura const para el esquema, es más usado y moderno que el Var.

Una vez que configuremos la base, ya tenemos todo configurado y listo para albergar los datos, sólo nos queda crear las rutas que definirán las llamadas a la API para poder guardar y consultar la información.

Conexión de la BD

```
const mongoose = require('mongoose'); // Importa la biblioteca Mongoose para MongoDB.

const connectDB = async () => { // Define una función asíncrona para conectar a la base de datos.
  try {
    // Intenta conectarse a MongoDB en la URL especificada.
    await mongoose.connect("mongodb://localhost:27017/tvshows", {
      // useNewUrlParser: true,
      // Habilita un nuevo analizador de URL (ahora es predeterminado).
      // useUnifiedTopology: true,
      // Activa la nueva capa de topología (ahora es predeterminado).
    });
    console.log('Conectado a MongoDB'); // Imprime un mensaje si la conexión es exitosa.
  } catch (err) {
    console.error('Error de conexión a MongoDB:', err); // Imprime el error si la conexión falla.
    process.exit(1); // Termina el proceso si no se puede conectar a la base de datos.
  }
};

module.exports = connectDB; // Exporta la función para que pueda ser utilizada en otros módulos.
```

Implementando los controladores de nuestras rutas o endpoints Obtener todas las series de tv (READ - GET)

```
const TVShow = require('../models/TVShow'); // Importa el modelo de TVShow.

// GET - Devuelve todas las series de TV en la base de datos
exports.findAllTVShows = async function (req, res) {
  try {
    const tvshows = await TVShow.find();
    // Espera a que se resuelva la promesa para obtener todas las series.
    console.log('GET /tvshows');
    res.status(200).json(tvshows); // Devuelve el resultado en formato JSON.
  } catch (err) {
    res.status(500).send(err.message); // Maneja errores y devuelve el mensaje.
  }
};
```

Los controladores de las rutas de nuestro API los vamos a crear en un archivo separado que llamaremos controllers/tvshows.js. Gracias a exports conseguimos modularizar y que pueda ser llamado desde el archivo principal de la aplicación.

Comparación del Módulo 9 con un código más actualizado

El segundo fragmento de código es más moderno, limpio y adherido a las mejores prácticas actuales en desarrollo de APIs con Node.js y Express. Utilizar async/await y manejar errores de manera consistente son claves para mantener la legibilidad y la robustez del código.

```
var mongoose = require('mongoose');
var TVShow = mongoose.model('TVShow');

//GET - Return all tvshows in the DB
exports.findAllTVShows = function (req, res) {
  TVShow.find(function (err, tvshows) {
    if (err) res.send(500, err.message);

    console.log('GET /tvshows');
    res.status(200).jsonp(tvshows);
  });
};
```

Código Original

```
const TVShow = require('../models/TVShow'); // Importa el modelo de TVShow.

// GET - Devuelve todas las series de TV en la base de datos
exports.findAllTVShows = async function (req, res) {
  try {
    const tvshows = await TVShow.find();
    // Espera a que se resuelva la promesa para obtener todas las series.
    console.log('GET /tvshows');
    res.status(200).json(tvshows); // Devuelve el resultado en formato JSON.
  } catch (err) {
    res.status(500).send(err.message); // Maneja errores y devuelve el mensaje.
  }
};
```

Código Mejorado

Comparación

Manejo de Promesas:

- **Original**: Usa un enfoque basado en callbacks con TVShow.find(function (err, tvshows) {...}), lo que puede llevar a un código más difícil de leer y manejar.
- **Mejorado**: Utiliza async/await, que simplifica la lectura y la estructura del código, facilitando el manejo de errores y el flujo de ejecución.

Manejo de Errores:

- **Original**: Utiliza res.send(500, err.message), lo cual es menos claro y no es el enfoque estándar en Express. Además, no se registra el error.
- **Mejorado**: Utiliza res.status(500).send(err.message) (aunque aún se podría mejorar enviando un objeto JSON para consistencia) y se recomienda registrar el error con console.error(err).

Consistencia en Respuestas:

- **Original**: Usa res.status(200).jsonp(tvshows), lo que es menos común hoy en día. La función jsonp() está diseñada para respuestas JSONP, que son menos utilizadas en aplicaciones modernas.
- **Mejorado**: Usa res.status(200).json(tvshows), que es la forma estándar de enviar respuestas JSON.

Declaración de Variables:

- **Original**: Usa var, que tiene un alcance de función y puede causar confusiones en el manejo de variables.
- **Mejorado**: Usa const, que es más seguro y promueve buenas prácticas al evitar reasignaciones accidentales.

Importación del Modelo:

- **Original**: Usa mongoose.model('TVShow') para obtener el modelo.
- **Mejorado**: Usa require('../models/TVShow'), lo que hace que sea más explícito y claro de dónde proviene el modelo.

Obtener una serie de tv (READ - GET)

```
// GET - Devuelve una serie de TV con el ID especificado
exports.findById = async function (req, res) {
  try {
    const tvshow = await TVShow.findById(req.params.id); // Busca la serie por ID.
    if (!tvshow) return res.status(404).send('TVShow no encontrado');
    // Maneja caso donde no se encuentra la serie.
    console.log('GET /tvshow/' + req.params.id);
    res.status(200).json(tvshow); // Devuelve la serie encontrada en formato JSON.
  } catch (err) {
    res.status(500).send(err.message); // Maneja errores y devuelve el mensaje.
  }
};
```

Este fragmento de código permite recuperar un documento específico de la colección TVShow.

La función findById es exportada y la utilizaremos cuando se reciba una petición GET en una ruta con un parámetro id: '/tvshow/:id'. ¿Como funciona?

La función utiliza el método findById() del modelo TVShow para buscar y recuperar un documento específico (una serie) de la colección TVShow de la base de datos.

El método toma dos parámetros: el primer parámetro es el identificador del documento a buscar, que obtenemos de req.params.id , el segundo parámetro es una función de callback que se ejecuta cuando la operación de búsqueda ha sido completada.

Al igual que en la función anterior, dentro de la función callback, si hay un error al buscar el documento, se envía una respuesta HTTP con código 500 y el mensaje de error. Si la búsqueda se realiza correctamente, se imprime un mensaje en la consola y se envía una respuesta HTTP con código 200 y el documento recuperado en formato JSON.

Crear una serie de tv (CREATE - POST)

```
// POST - Inserta una nueva serie de TV en la base de datos
exports.addTVShow = async function (req, res) {
  try {
    console.log('POST');
    console.log(req.body);
    const tvshow = new TVShow({
      // Crea una nueva instancia del modelo TVShow
      // con los datos del cuerpo de la solicitud.
      title: req.body.title,
      year: req.body.year,
      country: req.body.country,
      poster: req.body.poster,
      seasons: req.body.seasons,
      genre: req.body.genre,
      summary: req.body.summary,
    });
    const savedTVShow = await tvshow.save();
    // Guarda la nueva serie en la base de datos.
    res.status(201).json(savedTVShow);
    // Devuelve la serie creada en formato JSON.
  } catch (err) {
    res.status(500).send(err.message);
    // Maneja errores y devuelve el mensaje.
  }
};
```

Esta función se ejecutará cuando se reciba una petición POST en la ruta: '/tvshows'. La primera parte del código imprime en la consola el método HTTP y el cuerpo de la petición (req.body), para tener una idea de lo que se está recibiendo.