Contenidos a Trabajar

1. Introducción
2. Conexión a BBDD MySQL desde Node
3. Capa de Modelos
4. Capa de Servicios

Agencia de Aprendizaje a lo largo de la vida

Introducción

En el desarrollo de aplicaciones web, es común trabajar con bases de datos para almacenar y recuperar información. MySQL es un sistema de gestión de bases de datos relacional ampliamente utilizado que ofrece una variedad de funcionalidades y es compatible con varios lenguajes de programación. Node.js, junto con el framework Express, es una excelente combinación para crear aplicaciones web de manera eficiente y escalable.

En esta ocasión aprenderemos cómo establecer una conexión a una base de datos MySQL y organizar nuestro código utilizando capas de servicios y modelos en un programa Node.js con Express y MySQL.

Agencia de Aprendizaje a lo largo de la vida

Conexión a BBDD MySQL desde Node

Para interactuar con una base de datos MySQL desde Node.js, necesitamos un controlador o paquete que nos permita realizar operaciones de base de datos.

Un paquete comúnmente utilizado es "mysql2" que se puede instalar a través del administrador de paquetes NPM. Una vez instalado, podemos usarlo para establecer una conexión con nuestra base de datos MySQL utilizando los detalles de conexión, como el host, el puerto, el usuario, la contraseña y el nombre de la base de datos.

Veámoslo con el siguiente ejemplo:

```
const mysql = require('mysql2');

// Creamos un pool de conexiones

const pool = mysql.createPool({
  host: 'localhost',
  user: 'elpepe',
  password: 'admin123',
  database: 'latiendita',
  port: 3306,
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
});

module.exports = {
  conn: pool.promise()
};
```

Agencia de Aprendizaje a lo largo de la vida

- **const mysql = require('mysql2');:** Importa el módulo **mysql2**, que es una versión mejorada y compatible con Promesas del controlador MySQL para Node.js.

- **const pool = mysql.createPool({ ... });:** Crea un pool de conexiones utilizando la función **createPool** del módulo mysql2. El pool de conexiones permite administrar y reutilizar conexiones a la base de datos de manera eficiente. Los parámetros proporcionados en el objeto de configuración son los siguientes:

    - **host:** El nombre del host donde se encuentra la base de datos.
    - **user:** El nombre de usuario para autenticarse en la base de datos.
    - **password:** La clave para autenticarse en la base de datos.
    - **database:** El nombre de la base de datos a la que se desea conectar.
    - **port:** El número de puerto en el que se encuentra el servidor de la base de datos. En este caso, se establece como 3306, que es el puerto por defecto para MySQL.
    - **waitForConnections:** Indica si el pool de conexiones debe esperar cuando no hay conexiones disponibles y el límite de conexiones se ha alcanzado.
    - **connectionLimit:** El número máximo de conexiones que el pool puede tener a la vez. En este caso, se establece como 10.

- **queueLimit:** El número máximo de conexiones en espera en la cola del pool. En este caso, se establece como 0, lo que significa que no hay límite.

- **module.exports = { conn: pool.promise() };:** Exporta un objeto que contiene la propiedad **conn**, la cual es una promesa que representa una conexión del pool de conexiones. Al utilizar **.promise()** en el pool, se habilita el uso de promesas en lugar de callbacks para realizar consultas a la base de datos.

En resumen, este código crea un pool de conexiones a una base de datos MySQL utilizando **mysql2** y configura las opciones necesarias, como el host, el usuario, la contraseña, la base de datos, el puerto y los límites de conexiones. Luego, exporta una promesa que representa una conexión del pool, lo que facilita la ejecución de consultas y operaciones en la base de datos utilizando promesas en lugar de callbacks.

Capa de Modelos

La capa de modelos es responsable de definir la estructura y las operaciones relacionadas con los datos de nuestra aplicación. Aquí es donde definimos nuestros modelos de datos y las funciones para crear, leer, actualizar y eliminar registros en la base de datos. Los modelos nos permiten encapsular la lógica de acceso a la base de datos y abstraer los detalles de implementación.

Continuando con el ejemplo anterior, usemos nuestra conexión para crear las consultas a la base de datos:

```
const { conn } = require('../config/conn');

// Función para crear un nuevo usuario
async function createUser(user) {
  const query = 'INSERT INTO users SET ?';

  try {
    const [rows] = await conn.query(query, user);
    return rows;
  } catch (error) {
    throw error;
  } finally {
    conn.releaseConnection();
  }
}

module.exports = {
  createUser
};
```

El código anterior encapsula en un archivo dentro de la capa de modelos, la consulta a la base de datos para crear o insertar un nuevo usuario, utilizando la conexión **conn** proveniente de un archivo de configuración **(../config/conn)**.

1. **const { conn } = require('../config/conn');**: Importamos la conexión **conn** desde el archivo de configuración **../config/conn**.
2. **async function createUser(user) { ... }**: Definimos una función asincrónica **createUser** que acepta un parámetro **user**, que representa los datos del nuevo usuario a ser insertado en la base de datos.
3. **const query = 'INSERT INTO users SET ?';**: Creamos una consulta SQL de inserción para la tabla "users" utilizando la sintaxis **INSERT INTO ... SET**. El signo de interrogación (**?**) se utiliza para indicar que se reemplazará con los valores proporcionados en el siguiente parámetro.
4. **try { ... } catch (error) { ... } finally { ... }**: Usamos try-catch-finally para manejar los errores y liberar la conexión al finalizar.
5. **const [rows] = await conn.query(query, user);**: Ejecutamos la consulta SQL utilizando el método **query** de la conexión **conn**. Se utiliza **await** para esperar a que la consulta se resuelva. El resultado se desestructura para obtener el arreglo **rows** que contiene los registros afectados por la consulta.
6. **return rows;**: Retorna el resultado de la inserción, que son los registros afectados por la consulta.

7. **throw error;**: En caso de que ocurra un error durante la ejecución de la consulta, se lanza una excepción para que sea capturada por el bloque catch posterior.

8. **conn.releaseConnection();**: En el bloque finally, liberamos la conexión utilizando el método **releaseConnection()**. Esto garantiza que la conexión se devuelva al pool de conexiones, lo que permite que esté disponible para otras operaciones.

9. **module.exports = { createUser };**: Exportamos la función **createUser** como el único miembro del módulo para que pueda ser utilizado por otros módulos que lo importen.

En resumen, este código exporta una función **`createUser`** que utiliza la conexión **conn** para realizar una inserción de un nuevo usuario en la base de datos. Utiliza la sintaxis de promesas y manejo de errores para asegurar que la operación se ejecute de manera asincrónica y manejar cualquier excepción que pueda ocurrir. Al finalizar, libera la conexión para que pueda ser reutilizada por otras operaciones.

Capa de Servicios

La capa de servicios actúa como una interfaz entre las rutas de nuestra aplicación y la base de datos. Es responsable de procesar las solicitudes recibidas, realizar las operaciones necesarias en la base de datos y devolver los resultados correspondientes. Los servicios encapsulan la lógica de negocio y proporcionan una abstracción que nos permite separar las preocupaciones y mantener nuestro código modular y fácil de mantener.

Sigamos con nuestro ejemplo:

```
const users = require('../models/users');

const createUser = async (params) => {
  // Aqui iria alguna lógica adicional
  return users.createUser(params);
};

module.exports = {
  createUser
}
```

Este código actúa como una capa intermedia entre el código que invoca la creación de un nuevo usuario (modelo) y la capa que recibe la orden e información necesaria desde las rutas (controlador). Puede utilizarse para

agregar lógica adicional antes o después de llamar a la función **`createUser`** principal, como validaciones, transformaciones de datos u otras operaciones necesarias. Esto permite una mayor modularidad y flexibilidad en el manejo de la creación de usuarios en la aplicación.

De esta manera cerramos las capas lógicas (MODELOS y CONTROLADORES) de nuestra aplicación utilizando el patrón de arquitectura **MVC** y cómo interactúan entre sí gracias a capas adicionales como los servicios y las rutas.

El siguiente paso es trabajar con los archivos que retornaremos al cliente mediante la capa de VISTAS.

Agencia de Aprendizaje a lo largo de la vida