«Talento Tech»

Node JS

Clase 13

Buenos Aires Ciudad

Clase N° 13 - Modelo de datos y trabajo con JSON

Temario:

1. JSON
    - Parse
    - Stringify
2. Filesystem
3. Modelos

Objetivos de la Clase

En esta clase, nos enfocaremos en comprender y aplicar el concepto de la capa de Modelos en una API Rest. Además abordaremos cómo definir modelos que permitan organizar y manipular la información de manera eficiente. También, profundizaremos en el uso de JSON, un formato clave para el intercambio de datos en aplicaciones web. Exploraremos cómo convertir objetos JavaScript a JSON mediante JSON.stringify y cómo reconstruir objetos a partir de JSON usando JSON.parse, habilidades esenciales para trabajar con datos en una API. Finalmente, aprenderemos a acceder y manipular datos almacenados en archivos mediante Filesystem, lo que nos permitirá simular una base de datos simple y entender cómo los modelos interactúan con los datos persistentes.

JSON

JSON (JavaScript Object Notation) es un formato de texto ligero utilizado para el intercambio de datos. Su estructura es sencilla y fácil de leer para los desarrolladores, además de ser compatible con diversos lenguajes de programación. Es ampliamente empleado en aplicaciones web para la transmisión de información entre el cliente y el servidor, especialmente en API REST. Surgió como una alternativa a XML, que en su momento era el estándar predominante.

Un ejemplo de JSON sería el siguiente:

```json
{
  "name": "Juan",
  "age": 30,
  "isStudent": false,
  "courses": ["Matemáticas", "Programación"]
}
```

La sintaxis de JSON es similar a la de los objetos en JavaScript, con la diferencia de que las claves deben estar siempre entre comillas dobles (`""`) y no admite funciones como valores de sus propiedades.

Debido a esta diferencia, es necesario convertir los objetos entre ambos formatos en determinadas situaciones. Por ejemplo, al leer un archivo JSON en JavaScript, se requiere transformarlo en un objeto literal para manipularlo en el código. Del mismo modo, cuando se necesita guardar datos en un archivo JSON, es preciso convertir los objetos de JavaScript a este formato.

Para realizar estas conversiones, se utilizan los métodos `JSON.parse()` y `JSON.stringify()`, los cuales permiten transformar datos de JSON a objetos de JavaScript y viceversa, facilitando así el intercambio de información en aplicaciones web.

JSON.parse()

El método `JSON.parse()` se utiliza para convertir una cadena de texto en formato JSON a un objeto de JavaScript.

```
JSON.parse(stringJSON, reviver);
```

- `stringJSON`: Una cadena de texto en formato JSON válida.
- `reviver` *(opcional)*: Una función que puede modificar los valores antes de devolver el objeto final.

```
const jsonToConvert = {
  "name": "Juan",
  "age": 30,
  "isStudent": true,
  "courses": ["Matemáticas", "Programación"]
}

const data = JSON.parse(jsonToConvert);


console.log(data.name); // Juan
```

Uso del parámetro reviver

El parámetro reviver permite aplicar una transformación a los valores del objeto antes de que sea devuelto.

```
const data = JSON.parse(jsonToConvert, () => {
  return key === 'isStudent' ? true : key;
});
```

En el ejemplo anterior utilizamos una función de callback donde durante el proceso de conversión a objeto literal, cambia el valor de la key `isStudent` a `true`.

JSON.stringify()

El método `JSON.stringify()` convierte un objeto de JavaScript en una cadena de texto con formato JSON.

```
JSON.stringify(valor, replacer, espacio);
```

- `valor`: El objeto que se desea convertir en JSON.
- `replacer` *(opcional)*: Una función o un array que determina qué propiedades incluir en la conversión.
- `espacio` *(opcional)*: Un número o cadena que define la indentación en la salida para hacerla más legible.

```
const obj = { name: "Juan", age: 30, isStudent: false };
const jsonString = JSON.stringify(obj);

console.log(jsonString);

// '{"name":"Juan","age":30,"isStudent":false}'
```

Uso del parámetro espacio

Este parámetro permite generar una salida más legible agregando espaciados:

```
const obj = { name: "Luis", age: 40, city: "Madrid" };

const jsonString = JSON.stringify(obj, null, 2);
// Indentación de 2 espacios

console.log(jsonString);
/*
{
  "name": "Luis",
  "age": 40,
  "city": "Madrid"
}
*/
```

Estos métodos son fundamentales para el manejo de datos en aplicaciones web, facilitando el intercambio de información entre el servidor y el cliente.

Filesystem

El módulo **Filesystem (`fs`)** de Node.js permite interactuar con el sistema de archivos del servidor, proporcionando funciones para leer, escribir, modificar y eliminar archivos y directorios. Es una herramienta fundamental en el desarrollo backend, especialmente cuando se necesita manipular datos de manera persistente.

Importación del módulo `fs`

Para utilizar `fs`, es necesario importarlo en el archivo de código:

```
import fs from 'fs';
```

Node.js ofrece dos formas de trabajar con `fs`:

- **Modo síncrono**: Las operaciones bloquean la ejecución del código hasta que finalizan.
- **Modo asíncrono**: Las operaciones se ejecutan en segundo plano, sin bloquear el flujo del programa.

Lectura de archivos

Modo síncrono (`fs.readFileSync`)

```
import fs from 'fs';

const data = fs.readFileSync(data.txt', 'utf8');
console.log(data);
```

En este caso, la ejecución del código se detiene hasta que el archivo sea leído por completo.

`readFileSync` recibe 2 parámetros, el primero es la ruta a la ubicación de nuestro archivo dentro del servidor y el segundo es la codificación de caracteres, en este caso `'utf8'` con aceptación de caracteres Unicode.

Modo asíncrono (`fs.readFile`)

```
fs.readFile('data.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error al leer el archivo:', err);
    return;
  }
  console.log(data);
});
```

El archivo se lee en segundo plano y el resultado se devuelve a través de un callback.

`readFile` a diferencia del anterior, recibe 3 parámetros, los 2 primeros son iguales a `readFileSync` mientras que el tercero es una función del callback que se ejecutará una vez leído el archivo devolviendo su contenido o un error en caso de existir.

Otros métodos de `fs`

Además de métodos de lectura, `filesystem` posee otros que nos permiten escribir (modificar), eliminar y crear archivos e incluso directorios (carpetas) completos.

Algunos de ellos son:

- `fs.writeFileSync` y `fs.writeFile`: permite sobreescribir el contenido completo de un archivo.
- `fs.appendFileSync` y `fs.appendFile`: que agregan contenido a la última fila de un archivo.
- `fs.unlinkSync` y `fs.unlink`: que eliminan archivos.

Lectura asíncrona o síncrona

- Se recomienda **usar las versiones asíncronas** (`fs.readFile`, `fs.writeFile`, etc.) en aplicaciones web o servidores, ya que evitan bloqueos en la ejecución y mejoran el rendimiento.
- Las funciones **síncronas** (`fs.readFileSync`, `fs.writeFileSync`, etc.) son útiles en scripts pequeños o cuando es necesario garantizar que una tarea se complete antes de continuar.

***Nota:** en ocasiones el uso de **fs** se complementa con el módulo **path** y la variable global **__dirname** para poder obtener las rutas a los archivos de forma dinámica.

Recordemos que **__dirname** no está disponible directamente cuando utilizamos ESModules, por lo que con este pequeño snippet de código podemos resolverlo:

```
const __dirname = import.meta.dirname;
```

Modelos

En el contexto de una API Rest, los Modelos son representaciones de las entidades de datos que maneja la aplicación. Por ejemplo, si estamos construyendo una aplicación para gestionar usuarios, un modelo de Usuario definiría cómo se estructura la información de un usuario (nombre, correo, contraseña, etc.) y cómo se interactúa con esa información (crear, leer, actualizar o eliminar).

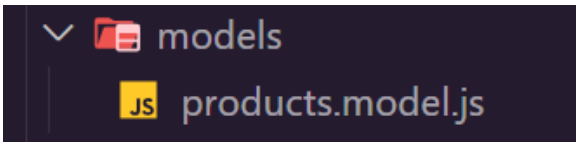
Los modelos actúan como una capa intermedia entre la lógica de negocio (servicios) y la fuente de datos (archivos, bases de datos, etc.). Su objetivo principal es encapsular la lógica relacionada con los datos, asegurando que la aplicación sea modular, mantenible y escalable.

Creación de un modelo en Node.js

Supongamos que estamos construyendo una API para gestionar productos. Primero, definiremos un modelo para representar un producto. En este caso, utilizaremos un archivo JSON como fuente de datos para simular una base de datos simple.

Creamos el modelo

Un modelo en Node.js puede ser una clase o un objeto que define las propiedades y métodos relacionados con los datos. Veamos un ejemplo de un modelo de Producto, para eso creamos nuestro archivo `products.model.js` dentro de nuestra carpeta `models`.



En nuestro archivo creamos un objeto con los métodos necesarios para manipular los datos provenientes en este caso de un archivo JSON:

```
// products.model.js
import fs from 'fs';
import path from 'path';

const __dirname = import.meta.dirname;
// Ruta al archivo JSON que simula la "base de datos"
const dataPath = path.join(__dirname, '../data/productos.json');
```

```
// Método para buscar un producto por su ID
export function getProductById(id) {
  const products = this.getAllProducts();
  return products.find(product => product.id === id);
};
// Método para obtener todos los productos
export function getAllProducts() {
  const data = fs.readFileSync(dataPath, 'utf-8');
  return JSON.parse(data);
};
// Método para guardar un producto en el archivo JSON
export function saveProduct(name, price) {
  const products = this.getAllProducts();
  products.push({ id: products.length, name, price });
  fs.writeFileSync(dataPath, JSON.stringify(products, null, 2));
};
// Método para eliminar un producto por su ID
export function deleteProduct(id) {
  const products = this.getAllProducts();
  products = products.filter(product => product.id !== id);
  fs.writeFileSync(dataPath, JSON.stringify(products, null, 2));
};
```

En el ejemplo anterior vemos un paralelismo marcado con nuestros archivos de `controller` y de `service` en la forma de declarar los métodos, sin embargo hemos logrado migrar la lógica de acceso a los datos a una nueva capa simulando las consultas a una base de datos pero en esta ocasión directamente trabajando son un archivo JSON gracias a la librería `fs(filesystem)` que vimos anteriormente.

Ahora nuestro archivo de `service` se ve de la siguiente manera:

```
// products.service.js
import * as productService from "../models/products.model.js";

export const getAllProducts = () => {
   return productService.getAllProducts();
};

export const getProductById = async (id) => {
  return productService.getProductById(id);
};

export const createProduct = async (productData) => {
  const { name, price } = productData;
```

```
  return productService.saveProduct(name, price);
  };

export const deleteProduct = async (id) => {
  return productService.deleteProduct(id);
};
```

Paralelismo entre Servicios y Modelos

En una arquitectura bien estructurada, los modelos y los servicios tienen responsabilidades claramente definidas, pero su interacción puede generar cierta confusión debido a su similitud en la declaración de métodos. Veamos cómo se relacionan:

Modelos

- Los modelos se encargan de interactuar directamente con los datos. En este caso, el modelo Product maneja la lectura y escritura del archivo JSON, simulando el acceso a una base de datos.
- Su objetivo es abstraer la lógica de acceso a los datos, proporcionando métodos como getAllProducts, getProductById, saveProduct y deleteProduct.
- Los modelos no deben contener lógica de negocio; su función es simplemente gestionar los datos.

Servicios

- Los servicios actúan como una capa intermedia entre los controladores y los modelos. Su objetivo es manejar la lógica de negocio y orquestar las operaciones necesarias para cumplir con los requisitos de la aplicación.
- En el ejemplo, el servicio products.service.js utiliza los métodos del modelo Product para obtener, crear o eliminar productos.
- Los servicios pueden enriquecer o transformar los datos antes de pasarlos al controlador. Por ejemplo, podrían combinar datos de múltiples modelos, aplicar reglas de negocio o validaciones adicionales.

Veamos algunos casos donde servicios son útiles frente a los modelos:

1. **Lógica de negocio:**

    - Los servicios permiten encapsular reglas de negocio que no pertenecen a la capa de modelos. Por ejemplo, si necesitas aplicar un descuento a todos los productos de una categoría específica, esta lógica debería estar en el servicio, no en el modelo.

```
export const applyDiscountToCategory = (category, discount) => {
    const products = Product.getAllProducts();
    const discountedProducts = products.map(product => {
      if (product.category === category) {
        return { ...product, price: product.price * (1 - discount) };
      }
      return product;
    });
    return discountedProducts;
};
```

2. **Combinación de datos:**

    - Si necesitas combinar datos de múltiples modelos, el servicio es el lugar ideal para hacerlo. Por ejemplo, podrías obtener información de productos y usuarios en una sola operación.

```
export const getProductWithUser = (productId, userId) => {
  const product = Product.getProductById(productId);
  const user = User.getUserById(userId);
  return { product, user };
};
```

3. **Validaciones adicionales:**

    - Los servicios pueden incluir validaciones que no son responsabilidad del modelo. Por ejemplo, verificar si un usuario tiene permisos para realizar una operación.

```
export const deleteProductIfAllowed = (productId, userId) => {
  const user = User.getUserById(userId);
  if (user.role === 'admin') {
    return Product.deleteProduct(productId);
  } else {
    throw new Error('Unauthorized');
  }
};
```

La separación entre **modelos** y **servicios** no es redundante, sino una práctica que promueve la modularidad y el principio de responsabilidad única. Los modelos se encargan exclusivamente de interactuar con los datos, mientras que los servicios manejan la lógica de negocio y orquestan operaciones más complejas. Esta división permite que el código sea más fácil de mantener, probar y escalar, especialmente en aplicaciones que crecen en complejidad.

Ejercicio Práctico

Organización de la Capa de Modelos en tu API

Sabrina y Matías regresan para revisar tu progreso. Esta vez, Sabrina es quien toma la palabra con entusiasmo:

"¡Tu organización ha mejorado muchísimo! Ya tienes rutas bien definidas, controladores claros y una capa de servicios eficiente. Ahora es momento de llevar la estructura un paso más allá."

Matías asiente y añade con una mirada seria:

"Hasta ahora, has estado manejando datos simulados directamente en la capa de servicios, pero en un proyecto real, los datos suelen almacenarse en bases de datos o archivos estructurados. Por eso, vamos a introducir la capa de modelos, que centralizará la gestión de datos."

Misión:

1. **Crear la capa de modelos**
    - Organiza los datos de tu aplicación en archivos JSON dentro de una carpeta llamada `data`.
    - Migra los datos simulados que estaban en los servicios hacia estos archivos.
    - Asegúrate de que la estructura de los archivos JSON sea clara y representativa de la información que manejas.
    - Crea archivos para los modelos de tu aplicación y crea los métodos necesarios para interactuar con los JSON de datos.

2. **Modificar los servicios para interactuar con los modelos**

    - Ajusta la capa de servicios para que en lugar de devolver datos directamente desde el código lo haga utilizando los métodos creados en los modelos.
    - Asegúrate que los controladores sigan obteniendo los datos correctamente desde los servicios.

Matías concluye con una mirada confiada:

"Si logras completar este desafío, habrás dado un paso gigante hacia el desarrollo backend profesional. ¡Es hora de escribir código limpio y estructurado!"

Materiales y Recursos Adicionales:

- **Documentación Oficial de JSON**: https://www.json.org/json-es.html
- **Documentación de Node.js sobre el módulo FS:** https://nodejs.org/api/fs.html

---

Preguntas para Reflexionar:

- ¿Cuál es la importancia de la capa de modelos en una API REST y cómo contribuye a la organización del código?
- ¿Qué ventajas tiene utilizar archivos JSON como base de datos en lugar de manejar datos directamente en los servicios?

- ¿Qué posibles problemas podrías encontrar al usar archivos JSON como almacenamiento de datos y cómo podrías solucionarlos?
- ¿Cómo podrías escalar la solución actual si en el futuro decides migrar a una base de datos relacional o NoSQL?

Próximos Pasos:

- **Datos en la nube**: Configurando y accediendo a datos en un servidor externo.
- **Autenticación y Autorización**: Manejando el acceso público y privado de nuestros datos.
- **Fin de la cursada**: daremos cierre al curso mediante la presentación de proyectos finales.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad