Anexos React con Typescript

Juan Carlos Pérez Rodríguez

Sumario

Anexo: json-server

Para poder desarrollar independientemente de una api se puede hacer uso de json-server, que es “fake server” con el que podemos interactuar como si estuviéramos trabajando con una api de verdad. De esa forma nos independizamos del desarrollo del lado del servidor y podemos hacer nuestra aplicación del lado cliente (ionic, react-native,…)

La documentación oficial y un buen tutorial lo encontramos en:
https://www.npmjs.com/package/json-server

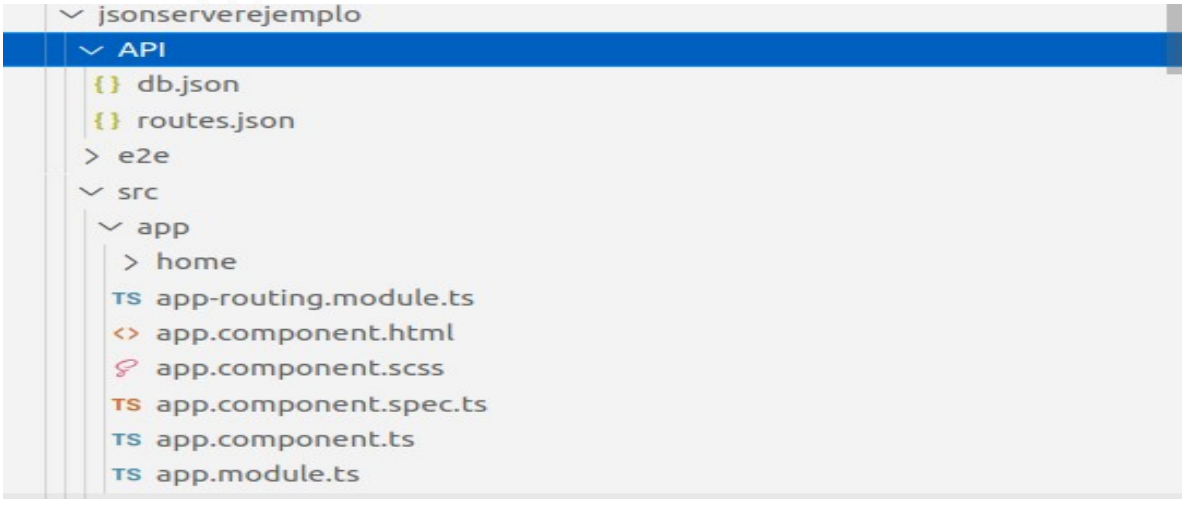
https://sigdeletras.com/2020/crear-una-fake-reat-api-con-json-server-copy/

Instalamos jsonserver (se ha elegido hacerlo de forma global: -g) mediante:

```
npm install  -g json-server
```

json-server recibe un fichero json con los datos que queremos mostrar en la api (en nuestro ejemplo lo llamaremos db.json) Como nosotros le especificamos donde está ubicado, lo podemos poner donde queramos. Ahora bien, como queremos tener la información organizada, parece lógico crear una carpeta llamada: API dentro de nuestro proyecto (react, angular, …) y poner ahí todo lo que precisemos para abrir el json-server

Para el caso de la aplicación monedas podemos crear una carpeta API en el proyecto y poner los ficheros: db.json (para los datos que quieras que devuelva el api) y routes.json (para las rutas personalizadas)



Nota: Si se usa json-server para una api normalmente da igual incorporar o no la parte de: /api/v1

Así json-server responde tanto a:
http://localhost:3000/api/v1/monedas

como a:
http://localhost:3000/monedas

sin embargo es preferible establecerlo en las rutas: --routes porque hay cosas que no funcionan del todo bien

Arrancar el servicio (por defecto se activa en el puerto 3000):

```
json-server --watch API/db.json --routes API/routes.json --id idmoneda
```

Observar **--id idmoneda** se ha "mapeado" lo que él espera encontrar como id: "id" a la etiqueta: "idmoneda". Esto es importante porque en las apis rest acostumbramos acceder con variables de path a los objetos. Así accedemos a la moneda con id 1 mediante: /api/monedas/1

Pero veamos que ocurre si tenemos el siguiente json:

```
{
  "idmoneda": 4,
  "nombre": "franco",
  "pais": "Francia",
  "historicos": []
},
```

El problema está en que, por defecto, json-server espera encontrar etiquetas llamadas: "**id**" para definir esos ids. Si nosotros hemos puesto otro nombre de identificador (en este caso el json escrito usa: **idmoneda**) no va a saber hacer bien las rutas que hemos dicho. Mediante el parámetro:

```
--id nombreid
```

conseguimos que lo haga correctamente

Así que la orden de inicio de json-server anterior ejecuta correctamente: /api/monedas/1
(mostrando la moneda con idmoneda=1)

Observar también: **--watch API/db.json** vemos que estamos especificando el fichero que va a hacer las veces de nuestra base de datos-api El parámetro: **--watch** es la forma que tenemos de decir que use determinado fichero json como base de datos. En este caso hemos puesto una ruta relativa: API/db.json porque estamos haciendo la presunción que estamos ejecutando json-server en una terminal que está dentro de nuestro proyecto (react, angular, …)

El fichero que hace de base de datos: db.json debe contener un único objeto json (abrimos llaves: "{" al comienzo del fichero y cerramos llaves justo al final del fichero: "}") Dentro de ese objeto podemos poner todos los otros objetos y arrays que queramos

Finalmente observar: **--routes API/routes.json** mediante ese parámetro: --routes estamos especificando el nombre del fichero para rutas personalizadas

Veamos un ejemplo de ese fichero:

```
{
	"/monedas/:id/historicos": "/historicos?idmoneda=:id"
}
```

Nosotros queremos que nos funcionen las rutas del tipo: "**api/monedas/3/historicos**" para especificar que nos muestre los objetos históricos relacionados para la moneda 3 (por ejemplo para GET obtener todos los cambios de moneda para la moneda 3 y se fuera POST sería agregar un nuevo histórico de cambio de moneda para la moneda 3)

Ahora veamos un trozo de como es el json que le hemos dado para monedas:

```
{
	"monedas": [
	  {
		"idmoneda": 1,
		"nombre": "dolar",
		"pais": "Estados Unidos",
		"historicos": [
		  {
			"idhistoricocambioeuro": 4,
			"fecha": "2019-12-27",
			"equivalenteeuro": 0.9
		  },
		  {
			"idhistoricocambioeuro": 5,
			"fecha": "2020-12-30",
			"equivalenteeuro": 0.95
		  }
		]
	  }
	]
}
```

Vemos que hay un array llamado: historicos que aparece en cada moneda para reflejar los diferentes tipos de cambio que ha habido para esa moneda.

Ahora un ejemplo de la parte de históricos que tenemos en ese fichero:

```
{

  "historicos":[
        {
          "idhistoricocambioeuro": 4,
          "fecha": "2019-12-27",
          "equivalenteeuro": 0.9,
          "idmoneda": 1
        },
        {
          "idhistoricocambioeuro": 5,
          "fecha": "2020-12-30",
          "equivalenteeuro": 0.95,
          "idmoneda": 1
        }


  ]
}
```

Como en los parámetros del json-server que iniciamos ya le hemos dicho cuál es el id para la moneda (idmoneda) el servidor ya sabe como manejar las rutas:

```
/monedas/:id
```

Adjuntar recursos estáticos (por ejemplo imágenes de acceso público)

Si estamos poniendo nuestro json-server en la misma carpeta donde está nuestro proyecto react, ponemos la carpeta de recursos estáticos (por ejemplo img) en public (realmente puede ser otro sitio si luego lo configuramos, pero es la ruta que se espera)

Lanzamos el comando estableciendo la ruta del json y de la carpeta para ficheros estáticos (img, css, js,...)

```
json-server --watch rutadondeestaeljson/peliculas.json --static rutadondeestaimg/img
```

Caso de rutas relacionadas (ejemplo: históricos relacionados con una moneda)

¿ como le indicamos la subruta historicos ? json-server tiene contempladas varias opciones de búsqueda.

El siguiente ejemplo es válido en algunos casos PERO NO PARA NUESTRO CASO: Existe por ejemplo: ?_embed=historicos **?_embed** permite encadenar una subruta que incluya, en este caso, historicos Así que la instrucción siguiente (si es capaz de identificar los ids de cada objeto json) genera objetos embebidos: pone el grupo de históricos que corresponde para la primera moneda dentro de esa moneda y lo muestra así:

```
api/monedas/1?_embed=historicos
```

Para lo dicho, lo que tenemos que hacer es "mapear" lo que nosotros queremos con lo que él sabe hacer. Volvamos a ver el fichero routes.json

```
{
    "/monedas/:id/historicos": "/monedas/:id?_embed=historicos"
}
```

Vemos que se le está diciendo que mapee las rutas del tipo que nosotros queremos (parte izquierda en color verde) con las que él sabe buscar (parte derecha en color azul)

PERO NO VAMOS A HACERLO ASÍ. En este caso no estamos trabajando con embebidos (estamos usando nombres de id diferentes de los que el utiliza y se complica un poco)

Veamos el que vamos a usar:

```
{
  "/api/v1/*": "/$1",
  "/monedas/:id/historicos": "/historicos?idmoneda=:id"

}
```

Primero nos dice que va a aceptar cualquier solicitud desde /api/v1 como si hiciéramos la llamada directamente (que es la forma que trabaja: directamente sobre recursos no como subruta)
Así existe: /api/v1/monedas y /api/v1/historicos y funciona igual que: /monedas, /historicos

La segunda línea dice: "/monedas/:id/historicos": "/historicos?idmoneda=:id" así que mapea las rutas: /monedas/2/historicos a: /historicos?idmoneda=2 Esto es: una búsqueda de los históricos que tienen idmoneda=2

Por defecto el servicio se ejecuta en el puerto 3000 para modificarlo se pasa el puerto. Ejemplo para 8080:

```
json-server --watch API/db.json --routes API/routes.json --id idmoneda --port 8080
```

Para cualquier otra opción acudir a la documentación oficial

Ejemplo completo de routes.json

```
{
  "/api/v1/*": "/$1",
  "/monedas/:id/historicos": "/historicos?idmoneda=:id"

}
```

La instrucción con la que arrancamos el servicio ya la hemos visto:

```
json-server --watch API/db.json --routes API/routes.json --id idmoneda --port 8080
```

Ejemplo completo de db.json:


```
{

  "historicos":[
        {
          "idhistoricocambioeuro": 4,
          "fecha": "2019-12-27",
          "equivalenteeuro": 0.9,
          "idmoneda": 1
        },
        {
          "idhistoricocambioeuro": 5,
          "fecha": "2020-12-30",
          "equivalenteeuro": 0.95,
          "idmoneda": 1
        },
        {
          "idhistoricocambioeuro": 6,
          "fecha": "2021-11-08",
          "equivalenteeuro": 2,
          "idmoneda": 1
        },
        {
          "idhistoricocambioeuro": 8,
          "fecha": "2021-12-16",
          "equivalenteeuro": 3,
          "idmoneda": 1
        },
        {
          "idhistoricocambioeuro": 2,
          "fecha": "2020-03-25",
          "equivalenteeuro": 0.25,
          "idmoneda": 2
        },
        {
          "idhistoricocambioeuro": 3,
          "fecha": "2019-01-14",
          "equivalenteeuro": 0.5,
          "idmoneda": 2
        }


  ],
  "monedas": [
    {
      "idmoneda": 1,
      "nombre": "dolar",
      "pais": "Estados Unidos",
      "historicos": [
        {
          "idhistoricocambioeuro": 4,
          "fecha": "2019-12-27",
          "equivalenteeuro": 0.9
        },
        {
          "idhistoricocambioeuro": 5,
          "fecha": "2020-12-30",
          "equivalenteeuro": 0.95
```

```
        },
        {
          "idhistoricocambioeuro": 6,
          "fecha": "2021-11-08",
          "equivalenteeuro": 2
        },
        {
          "idhistoricocambioeuro": 8,
          "fecha": "2021-12-16",
          "equivalenteeuro": 3
        }
      ]
    },
    {
      "idmoneda": 2,
      "nombre": "Yen",
      "pais": "Japan",
      "historicos": [
        {
          "idhistoricocambioeuro": 2,
          "fecha": "2020-03-25",
          "equivalenteeuro": 0.25
        },
        {
          "idhistoricocambioeuro": 3,
          "fecha": "2019-01-14",
          "equivalenteeuro": 0.5
        }
      ]
    },
    {
      "idmoneda": 4,
      "nombre": "franco",
      "pais": "Francia",
      "historicos": []
    },
    {
      "idmoneda": 5,
      "nombre": "libra",
      "pais": "UK",
      "historicos": []
    },
    {
      "idmoneda": 6,
      "nombre": "dolar",
      "pais": "Australia",
      "historicos": []
    },
    {
      "idmoneda": 7,
      "nombre": "dolar",
      "pais": "Canadá",
      "historicos": []
    }
  ]
}
```

Anexo: Bases para Typescript

Para crear un proyecto react que soporte typescript:

npx create-react-app comienzo-react --template typescript

Definición de typescript

TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipos estáticos y objetos basados en clases (fuente wikipedia)

Tipos en Typescript

La principal característica de Typescript es su sistema de tipos, el cual realiza una formalización de los tipos de Javascript, mediante una representación estática de su sistema de tipado dinámico. Esto permite a los desarrolladores definir variables y funciones fuertemente tipadas sin perder la esencia de Javascript (su naturaleza debilmente tipada y su extremada flexibilidad). Poder definir los tipos durante el tiempo de diseño nos ayuda a evitar errores en tiempo de ejecución, como podría ser pasar el tipo de variable incorrecto a una función.

Javascript ya viene con soporte de **String** y **Number** . Así que tiene:

- **Array**: tipo de dato estructurado que permite almacenar una colección de elementos.
  Así el tipo: Array<string> es un array de string (cadenas de texto)
- **String**: cadenas de texto.
- **Number**: números (tanto enteros como reales)
- **Boolean**: tipo de dato logico que representa verdadero o falso.
- **Enum**: representa al tipo enumeración. Una enumeración es una forma de dar nombres descriptivos a los conjuntos de valores numéricos
- **Any**: indica que la variable puede ser de cualquier tipo. Es muy útil a la hora de trabajar con librerías externas.
- **Void**: indica que una función no devolverá ningún valor.

Adicionalmente, aunque de menor uso:

- **Tuple**: similar al array, pero con un número fijo de elementos escritos.
- **Never**: este tipo representa el tipo de valores que nunca se producen. Por ejemplo para indicar que una función siempre arroja una excepción o que nunca termina su ejecución.

Adicionalmente se pueden crear agrupaciones de datos (estructuras de datos) como un tipo de dato personalizado mediante la instrucción: type o **interface**

Declaración de variables

Typescript declara las variables de la misma forma que javascript. Usando las palabras clave: let, var, const

- **const** para valores constantes

- **var** para variables con ámbito la función donde esté definida

- **let** para variables con ámbito las llaves donde ha sido definida

Veamos una declaración con let, que será la más usual:

```
let miNombre: string = “Juan San Epifanio”;
```

vemos que se especifica el tipo mediante el símbolo dos puntos: ":" después del nombre de la variable y justo después el tipo. En este caso estamos declarando la variable: miNombre y vemos que es de tipo string. Opcionalmente se puede hacer una asignación inicial después de la declaración mediante el igual: "="

Tener dos o más tipos en una variable

Para que una variable soporte varios tipos lo hacemos usando el pipe: "|" Así la siguiente declaración permite que la variable nombre pueda ser de tipo texto o de tipo número

let nombre: string | number ;

Declaración de tipos en las funciones

De una forma similar a antes, vemos en el siguiente ejemplo que los parámetros recibidos se define su tipo mediante el símbolo dos puntos: ":" después del nombre de la variable y justo después el tipo de dato esperado. Si se quiere especificar lo que devuelve se hace después de los paréntesis y antes del comienzo del cuerpo de la función (antes de las llaves). En el ejemplo se reciben dos números y se devuelve un texto:

```
sumar ( num1:number, num2:number) :string  {
        return "la suma resultante es: " + (a + b);
}
```

Interface

En typescript tenemos interfaces (no en javascript)

estas interfaces son patrones parecidos a las clases (aceptar atributos) pero **NO** hay instancias. Se puede asimilar a que son una regla de validación para objetos literales

En el siguiente ejemplo creamos un interfaz Moneda que coincida correctamente con los objetos que devuelva el json:

```
interface Moneda {
        idmoneda: number;
        nombre: string;
        pais: string;
        historicos: Historico[];
}

interface Historico {
        idhistoricocambioeuro: number;
        fecha: string;
        equivalenteeuro: number;
}
```

Así ahora se puede declarar una variable que siga ese patrón:

```
let miMoneda: Moneda;
```

Módulos en Typescript

Otra de las características de Typescript es heredada de ECMAScript la posibilidad de crear módulos, loc cuales no son más que una forma de encapsular código en su propio ámbito. Nos permiten agrupar nuestro código en diferentes ficheros, permitiéndonos exportarlos y utilizarlos donde los necesitemos. Esto nos facilita la tarea de crear software más ordenado. Así para que algún trozo de código o elemento esté disponible fuera del fichero usamos la palabra clave: **export**

```
export interface StringValidator {
  isAcceptable(s: string): boolean;
}
```

```
Ahora desde otro fichero:
import { StringValidator } from "./StringValidator";
```

```
}
```

Clases en Typescript

Las clases en typescript tienen mucha coincidencia con javascript. Por ejemplo, el constructor se define mediante la palabra: constructor() Y aunque tiene cierto soporte para la sobrecarga de constructores realmente trabaja con uno únicamente. Nosotros usaremos un único constructor

Por lo demás no es muy diferente de otros lenguajes orientados a objetos: se definen los atributos y se establece su visibilidad (public, private) Veamos un ejemplo en la documentación oficial:

```
class GoodGreeter {
  name: string;

  constructor() {
    this.name = "hello";
  }
}
```

Importante observar que SE DEBE ESCRIBIR SIEMPRE: **`this`** delante del atributo para usarlo en cualquier otra parte de la clase (salvo en su declaración).

Visibilidad de los atributos y métodos

Especificamos la visibilidad con: **public, private y protected** como en otros lenguajes

Únicamente tener en cuenta que la visibilidad por defecto es: **public**

Herencia

No hay diferencia respecto a otros lenguajes de programación en el tratamiento de la herencia. Hacemos uso de la palabra reservada: **extends**