

API Rest

Desarrollo Web Cliente.



API Rest

API Rest	2
Principios de una API REST	2
JSON server.....	3
Cliente	4
GET.....	4
POST	4
DELETE.....	5
PUT	5
Códigos de estado.....	5
1XX(Informativos)	5
2XX (Éxito)	5
3XX (Redirección).....	6
4XX Errores del cliente	6
5XX Errores del servidor	6

API Rest

Una **API REST** (Interfaz de Programación de Aplicaciones basada en Transferencia de Estado Representacional) es un estilo de arquitectura para crear servicios web que permiten a las aplicaciones interactuar entre sí a través de internet. Este tipo de API utiliza:

- HTTP para realizar la comunicación entre cliente y servidor
- REST como solución de arquitectura para el intercambio de recursos (utilizando para la comunicación HTTP)

Principios de una API REST

Para que una API se considere RESTful, debe cumplir con ciertos principios:

1. **Cliente-Servidor:** La iniciativa de la petición siempre la tiene el cliente.

2. **Sin Estado:** Cada solicitud del cliente al servidor debe incluir toda la información necesaria para entender y procesar la solicitud. Esto significa que el servidor no mantiene ningún estado de la sesión del cliente entre solicitudes.
3. **Interfaz Uniforme:** La API debe tener una interfaz consistente, lo que significa que los recursos y operaciones deben ser fácilmente entendibles y predecibles. Esto generalmente implica el uso de los métodos HTTP estándar:
 - a. **GET** para obtener recursos.
 - b. **POST** para crear nuevos recursos.
 - c. **PUT** o **PATCH** para actualizar recursos.
 - d. **DELETE** para eliminar recursos.
4. **Representación de Recursos:** Los datos (o recursos) son representados de forma estándar, generalmente en JSON. La API debe poder enviar, recibir y manipular recursos en base a las solicitudes de los clientes.

JSON server

Para poder hacer pruebas relativamente sencillas vamos a hacer uso de JSON server, que va a emular lo que sería un backend real, en este caso con las mínimas prestaciones, pero suficiente para ver el envío y recepción de datos.

Ten en cuenta que esto implica que vamos a tener dos proyectos ejecutándose a la vez, en este caso, los dos en localhost, pero en puertos diferentes.

Instalación:

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

Creación del fichero base:

En el directorio desde el que vamos a ejecutar el servidor tenemos que crear un fichero denominado *db.json*. Ahí estará lo que sería nuestra base de datos. Para las pruebas vamos a utilizar el siguiente contenido:

```
{
  "users": [
    {
      "id": 1,
      "name": "John Smith",
      "email": "john.smith@example.com",
      "age": 25
    },
    {
      "id": 2,
      "name": "Emma Johnson",
      "email": "emma.johnson@example.com",
      "age": 30
    },
    {
      "id": 3,
      "name": "Michael Brown",
      "email": "michael.brown@example.com",
      "age": 28
    },
    {
      "id": 4,
      "name": "Olivia Williams",
      "email": "olivia.williams@example.com",
      "age": 22
    },
    {
      "id": 5,
      "name": "James Jones",
      "email": "james.jones@example.com",
      "age": 35
    }
  ]
}
```

```
  ]}
```

Arranque

```
json-server --watch db.json
```

Si el servidor ha arrancado correctamente deberían aparecer la lista de endpoints. En nuestro caso únicamente el de users, ejecutándose en <http://localhost:3000/users>.

Cliente

Vamos a utilizar una aplicación cliente básica que permita recuperar los datos de los clientes, añadir uno nuevo y modificar o eliminar uno existente. En esta documentación únicamente vendrán definidas las distintas llamadas fetch.

GET

Fetch utiliza el método get por defecto, con lo que no es necesario indicar nada más que la url. Es la llamada que hemos estado utilizando hasta ahora. Vamos a probar desde el front-end que accedemos a los datos del servidor, para ello deberemos ejecutar:

```
fetch("http://localhost:3000/users")  
  .then(res => res.json())  
  .then(data => setUsers(data))
```

POST

Para añadir un recurso, en este caso un usuario hay que indicar el método, que será post, y el body, que serán los datos. Antes de enviar los datos hay que convertirlos a objeto JSON:

```
fetch("http://localhost:3000/users", {  
  method: 'POST',  
  body: JSON.stringify(data)  
}).then(res => {if (res.ok) getUsers()})
```

La petición se ha ejecutado correctamente si devuelve a true el campo ok. También se puede comprobar el campo status y los posibles errores.

DELETE

Para eliminar un recurso utilizamos el método delete pasando el id del recurso a borrar:

```
fetch(`http://localhost:3000/users/${id}`,  
  {method: 'DELETE'}  
)
```

PUT

Para modificar un recurso ya existente se utiliza el método put. Por un lado necesita el id como en la llamada al método delete y por otro lado los datos como hacíamos en el post. Este método modifica todo el recurso con los nuevos valores. El id no se debe modificar.

```
fetch(`http://localhost:3000/users/${data.id}`, {  
  method: 'PUT',  
  body: JSON.stringify(data)  
})
```

En el ejemplo anterior data es un objeto con los nuevos datos del recurso usuario a modificar.

Códigos de estado

Toda respuesta a una petición HTTP contiene un campo denominado status en el que vienen indicados los estados de error y éxito. Hay que tener en cuenta que fetch no devuelve error en la promesa a no ser que no haya sido capaz de llegar al servidor. Por eso no es suficiente con hacer un catch, aunque sí es necesario para determinar esos errores. El primer número del código de estado es el que da más información, siendo los siguientes los principales:

1XX(Informativos)

2XX (Éxito)

Los más utilizados son

200-Todo correcto

201- Recurso creado (respuesta de un post)

3XX (Redirección)

301-Movido permanentemente

4XX Errores del cliente

400 –Bad request

401-Unauthorized

403-Forbidden

404-Not found

405-Not allowed

5XX Errores del servidor

500-Internal Server Error