Clientes REST en JavaScript

Miguel Ortuño
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

Diciembre de 2023

Introducción a los clientes REST en JavaScript

Hasta ahora hemos visto

- Cómo ejecutar programas JavaScript dentro del navegador
- Cómo servir programas y ficheros con interface REST/ROA en el servidor, usando Express.js

A continuación, veremos cómo programar un cliente REST/ROA en JavaScript dentro del navegador. Lo probaremos con un servidor en Express.js

Same-origin policy

Same-origin policy es una norma que aparece en Netscape 2 (año 1995), que se ha convertido en un estándar. Consiste en que el código JavaScript solo puede acceder a datos que provengan del mismo origen desde el que se ha cargado el script

- Se entiende por *origen* el mismo protocolo, máquina y puerto

Ejemplo

- El usuario accede a una página web en *molamazo.com*,
- Esta página web puede tener código JavaScript que acceda a datos que estén en *molamazo.com*, pero solamente en este sitio
- No puede acceder a datos en *bancofuenla.es*
  - De lo contrario, una vez que el usuario se autentica en *bancofuenla* con una página de *bancofuenla*, un script malicioso en *molamazo* podría acceder a información sensible en *bancofuenla*

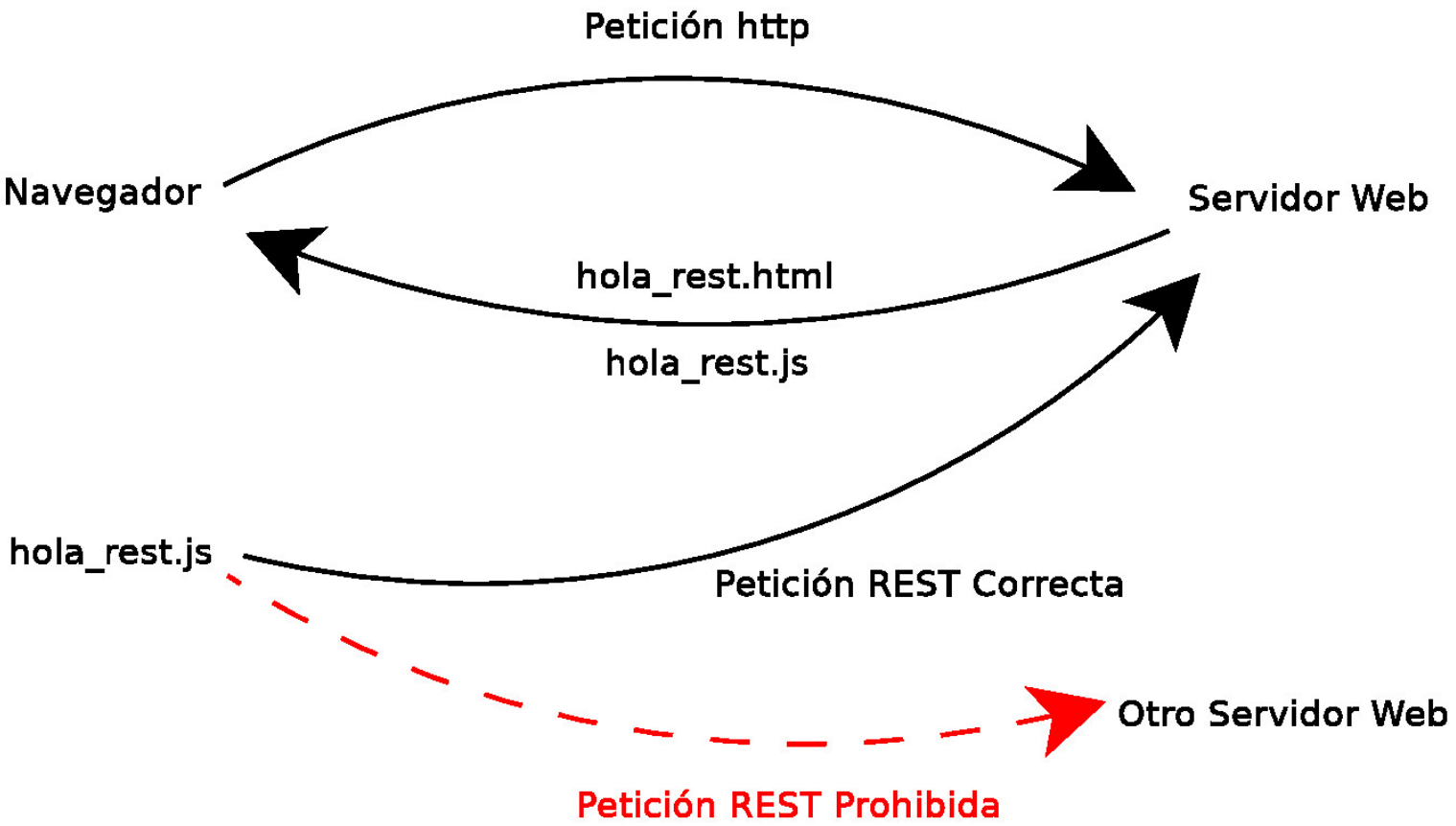

Figura: Same-Origin Policy

En ocasiones la *same-origin policy* resulta demasiado estricta y se requieren de técnicas que permitan, con el control adecuado, que un cliente haga peticiones a ciertos *orígenes* distintos

- JSONP (JSON with padding, JSON con relleno). Protocolo del año 2005. Obsoleto
- CORS. (Cross-Origin Resource Sharing). Protocolo desarrollado en 2004 y aceptado por el W3C en 2014

Aquí no lo veremos cómo aplicarlos, así que nuestros clientes REST estarán obligados a consultar con un servidor REST en el mismo origen (lo que en general es la opción preferible, la más sencilla y segura)

En la configuración que veremos aquí como ejemplo:

- Prepararemos un servidor con Express.js para que acepte peticiones REST/ROA en el puerto 3000 de *localhost*
- Por tanto, por la *same-origin policy*, el código JavaScript donde esté el cliente de este servidor, necesariamente debe haberse servidor por el servidor web en el puerto 3000 de *localhost*
- Por tanto, además de configurar Express.js para aceptar peticiones REST/ROA, también debemos configurarlo para servir los ficheros HTMl y JavaScript del cliente

En los ejemplos vistos en la asignatura hasta ahora, el navegador leía los ficheros HTML y los ficheros JavaScript directamente del sistema de ficheros local (el disco duro). Pero si siguiéramos trabajando así, incumpliríamos la *same-origin policy* (y no funcionaría a menos que configuráramos CORS)

Promesas

El uso de *promesas* es una técnica de programación concurrente disponible en lenguajes de programación como Java, JavaScript 6, C++, C#, Scala y muchos otros

- Una *promesa* es un objeto que inicialmente no tiene valor, pero que transcurrido algún tiempo, si todo va bien tendrá valor. O si no, lanzará una excepción
- En ECMAScript 2017 están disponible mediante las palabras reservadas *async await* (aunque hay otra sintaxis un poco más farragosas y antiguas)

Para extraer el valor de una promesa, anteponemos el operador *await*. Esto hace que la función se quede bloqueada esperando por el valor.

```
let respuesta =  await fetch(url);
```

El operador *await* solo se puede usar en funciones que hayan sido declaradas anteponiendo la palabra reservada *async*. Esto hace que la función sea asíncrona. También convierte el valor devuelvo por la función en una promesa

```
async function trae_resultado (numero) {
    [...]
    let respuesta =  await fetch(url);
    [...]
}
```

Aquí usaramos las promesas en la función *fetch()* que sirve para que un programa en JavaScript haga una petición HTTP, típicamente una petición REST/ROA

- A la función *fetch()* le pasamos una URL como argumento y devuelve una promesa con el resultado de la petición.
- Ese resultado es un objeto *Response* que contiene la respuesta HTTP en bruto, normalmente al programador lo que le interesará es el contenido en json. Para ello, el objeto *Response* dispone el método *json()* que devuelve una segunda promesa, con el objeto json.

```
async function trae_resultado (numero) {
  let dir_base = "http://localhost:3000/"

  let recurso = "api/dobla/";
  let url = dir_base + recurso + numero;
  console.log("url: " + url);

  let respuesta =  await fetch(url);
  console.log("respuesta: " + respuesta);

  let respuesta_json =  await respuesta.json();
  console.log("Respuesta_json: "+ respuesta_json);

  return respuesta_json;
}
```

- Si olvidamos el *await*, al trazar el valor, veremos que no contiene el json que esperaríamos, sino un objeto *promise*
- Si olvidamos añadir *async* a la función que usa *await*, obtendremos una excepción *await is only valid in async functions and the top level bodies of modules*

Recuerda: añadir *async* convierte el valor devuelto por la función en promesa. Por tanto, para leerlo, habrá que poner a otro *await*, que a su vez puede requerir otro *async*, que requerirá otro *await*... así hasta la función de más alto nivel del programa

- En otras palabras: cuando una función use una promesa, es necesario que esa función sea asincrona, y también su función *padre* (la función que la llama) y su abuelo, bisabuelo, tatarabuelo...

```
1   "use strict"
2  async function trae_resultado (numero) {
3    let dir_base = "http://localhost:3000/"
4
5    let recurso = "api/dobla/";
6    let url = dir_base + recurso + numero;
7
8    let respuesta =  await fetch(url);
9    let respuesta_json =  await respuesta.json();
10   return respuesta_json;
11
12 async function manej_boton01(event) {
13   //Número entero aleatorio entre 0 y 99
14   let numero = Math.floor(100 * Math.random());
15
16   span01.textContent = numero;
17   span02.textContent = await trae_resultado(numero);
18 }
19
20 let boton01 = document.querySelector("#boton01");
21 boton01.addEventListener("click", manej_boton01);
```

- Línea 8. Como *fetch()* devuelve una promesa, hay que anteponer *await*
- Línea 9. Como *json()* devuelve una promesa, hay que anteponer *await*
- Línea 2. Como en las líneas 8 y 9 hay un *await*, hay que anteponer *async*. Esto provoca que *trae_resultado* devuelva una promesa
- Línea 17. Como *trae_resultado* devuelve una promesa, hay que anteponer *await*
- Línea 12. Como en la línea 17 hay un *await*, hay que anteponer *async*

La función *manej_boton01* ya es de nivel global, no es necesario nigún *await* más (no se usa al registrar el manejador con *addEventListener* pero que esto no lee el valor devuelto por la función, solo lo vincula con el evento)

Ejemplo completo: doble de un número

En este ejemplo, tenemos un servidor REST/ROA que recibirá como parámetro un número y devolverá el doble de ese número

- Dirección base:
  `http://localhost:3000`
- Recurso:
  `/api/dobla/`
- Parámetro: `num`
- URL Completa:
  `http://localhost:3000/api/dobla/:num`

Ficheros requeridos

Necesitamos los siguientes ficheros:

- dobla_client.html
  Página web con el cliente
- dobla_client.js
  Código JavaScript del cliente
- dobla_server.js
  Script que configura Express para servir tanto las peticiones REST como los ficheros del cliente

```
http://localhost:3000/api/dobla/:num
http://localhost:3000/dobla_client.html
http://localhost:3000/js/dobla_client.js
```

Para probarlo en tu ordenador tendrás que:

1) Crear el directorio raiz para Express

   `~/www/site01/`

2) Copiar

   `dobla_client.html` a `~/www/site01/`

   `dobla_client.js` a `~/www/site01/js`

3) Lanzar Express.js con el fichero de configuración

   `node dobla_server.js`

4) Comprobar que el servidor atiende a las peticiones REST, pidiendo con tu navegador una página como p.e. `http://localhost:3000/api/dobla/10`
5) (opcional)
   Comprobar que Express sirve el código JavaScript, solicitando con el navegador la página
   `http://localhost:3000/js/dobla_client.js`
6) Comprobar que Express sirve la página web y que esta funciona correctamente
   `http://localhost:3000/dobla_client.html`

dobla_client.html

```html
<!DOCTYPE html>
<html lang="es-ES">

<head>
  <meta charset="utf-8">
  <title>Cómo enviar peticiones REST</title>
</head>

<body>
  <button id="boton01">Enviar un valor </button>
  <br>
  Valor enviado: <span id="span01"></span>
  <br>
  Valor recibido: <span id="span02"></span>

  <script src="js/dobla_client.js">
  </script>
</body>

</html>
```

http://ortuno.es/dobla_client.html

dobla_client.js

```
 "use strict"
async function trae_resultado (numero) {
  let dir_base = "http://localhost:3000/"

  let recurso = "api/dobla/";
  let url = dir_base + recurso + numero;
  console.log("url: " + url);

  let respuesta =  await fetch(url);
  console.log("respuesta: " + respuesta);
  let respuesta_json =  await respuesta.json();
  console.log("Respuesta_json: "+ respuesta_json);
  return respuesta_json;
}
```

```
async function manej_boton01(event) {
  //Número entero aleatorio entre 0 y 99
  let numero = Math.floor(100 * Math.random());

  span01.textContent = numero;
  span02.textContent = await trae_resultado(numero);
}

let boton01 = document.querySelector("#boton01");
boton01.addEventListener("click", manej_boton01);
```

http://ortuno.es/js/dobla_client.js.html

Atención en la construcción del path:

- Si lo generamos concatenando cadenas, hay que tener cuidado de que separando dirección base, recurso y argumentos haya exactamente una barra
  Es muy fácil cometer errores como

  `http://localhost:3000/api/dobla//:num`

  o

  `http://localhost:3000api/dobla/:num`

- Para evitarlos, en vez de concatenar las cadenas, podemos usar el método `path.join()` que se encarga de unir los fragmentos de URL, garantizado que haya una barra y solo una barra

```
path.join( dir_base, recurso, num );
```

dobla_server.js

```
const express = require('express');
const app = express();
const puerto = process.env.PORT || 3000;

app.use(express.json());
// middleware necesario para procesar las peticiones
// POST que incluyan un cuerpo json

// Direccionamiento estático:
const path = require('path');
// Importamos el módulo path

dir_raiz = path.join( process.env.HOME, "www/site01");
// Construye ~/www/site01

app.use(express.static(dir_raiz))
// Sirve todos los ficheros del directorio raiz
```

```
// Direccionamiento para GET
app.get('/api/dobla/:num', (req, res) => {
  let num = req.params.num;
  doblado = num * 2;
  // Normalmente aquí iría llamada a función en otro fichero

  console.log("El cliente envía " + num);
  console.log("La respuesta es " + doblado);
  doblado_json = JSON.stringify(doblado);
  res.json(doblado_json);
})
```

```
// Error 404
app.use((req, res) => {
  res.type('text/plain');
  res.status(404);
  res.send('404 - Dirección no encontrada');
})

// Error 500
app.use((err, req, res, next) => {
  console.error(err.message);
  res.type('text/plain');
  res.status(500);
  res.send('500 - Error en el servidor');
})

app.listen(puerto, () => console.log(
  `Express iniciado en http://localhost:${puerto}\n` +
  `Ctrl-C para finalizar.`));
```

http://ortuno.es/dobla_server.js.html

Captura del Status Code

Para leer el *status code* en el cliente, solo tenemos que leer la propiedad *status* del objeto promesa con la respuesta

- Si el código no es 200, significa que ha habido problemas y por tanto no podemos extraer ningún objeto json

```
async function trae_resultado (numero) {
  [...  obtiene la URL ...]

  let respuesta_bruto =  await fetch(url);
  let respuesta;  // Valor que devolverá esta función

  if (respuesta_bruto.status === 200) {
    respuesta = await respuesta_bruto.json();
    console.log("Respuesta correcta: "+ respuesta);
  else {
    respuesta = "Error " + respuesta_bruto.status;
    console.log(respuesta);}
  }

  return respuesta;
}
```