Desarrollo de Front-END

Tema 12

Clientes REST en JavaScript

César Andrés Sánchez

Universidad Camilo José Cela
Escuela Politécnica Superior de Tecnología y Ciencia

Curso 2025 - 2026

Clientes REST en JavaScript

¿Qué vamos a aprender?

En esta sección veremos cómo un programa JavaScript que se ejecuta en el navegador puede comunicarse con un servidor usando peticiones REST.
Hasta ahora ya aprendimos:

- Cómo ejecutar programas JavaScript dentro del navegador.
- Cómo crear un servidor con Express.js que exponga interfaces REST/ROA.

Ahora aprenderemos:

- Cómo programar un cliente REST en JavaScript.
- Cómo hacer peticiones HTTP usando la función `fetch()`.
- Cómo interpretar la respuesta del servidor (generalmente en formato JSON).

Trabajaremos con un servidor hecho con Express.js.

Same-Origin Policy (explicación sencilla)

La *Same-Origin Policy* es una norma creada para evitar accesos inseguros entre sitios web. Indica que el código JavaScript sólo puede acceder a datos del mismo origen desde el que fue cargado.

¿Qué es "el mismo origen"?

Dos direcciones tienen el mismo origen si coinciden:

- El protocolo (http / https).
- El dominio o IP.
- El puerto.

Ejemplo:

- Si visitas `molamazo.com`, su JavaScript sólo podrá pedir datos a `molamazo.com`.
- No podrá acceder a datos de `bancofuenla.es`.

Sin esta política, un sitio malicioso podría acceder a tus datos privados una vez iniciada sesión en otro sitio.

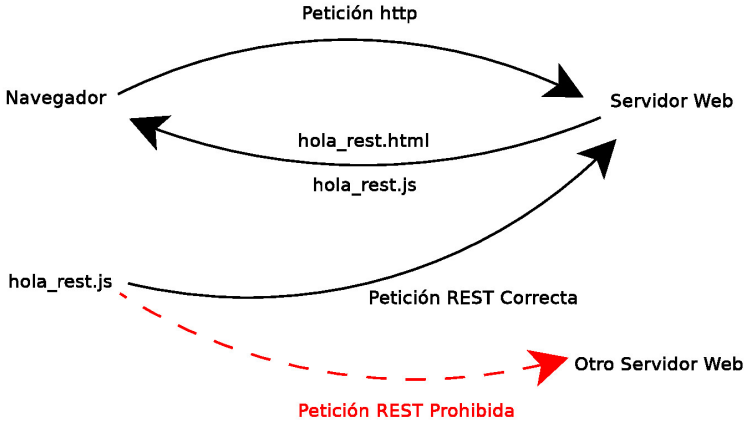

Figura: Ejemplo gráfico de la Same-Origin Policy

Cuando la política Same-Origin es demasiado estricta

A veces necesitamos acceder a recursos situados en otros orígenes. Para ello existen mecanismos que lo permiten de forma controlada:

- **JSONP**: técnica antigua (2005), actualmente obsoleta.
- **CORS**: mecanismo moderno (W3C 2014) para habilitar accesos entre orígenes.

En este tema NO aplicaremos CORS ni JSONP. Por tanto, nuestros clientes REST deberán comunicarse con el servidor REST en el mismo origen.

Configuración usada en los ejemplos

En los ejemplos:

- El servidor Express se ejecutará en `http://localhost:3000`.
- El navegador debe cargar el cliente (HTML y JavaScript) desde el mismo origen.
- Por la Same-Origin Policy, el cliente DEBE servirse desde el mismo servidor Express.

Si abrimos un fichero HTML directamente desde el disco local (`file:///...`), la política de origen impediría las peticiones REST, salvo que configuráramos CORS.

Promesas: explicado muy sencillo

Una *promesa* es un objeto que representa un valor que todavía no está disponible, pero lo estará en el futuro.
Las promesas se usan cuando:

- Una operación tarda un tiempo (por ejemplo, una petición HTTP).
- Puede completarse correctamente o fallar.

En JavaScript moderno (ECMAScript 2017) se usan las palabras clave:

- `async` para indicar que una función devuelve una promesa.
- `await` para esperar el resultado de una promesa.

Uso de async y await

Si queremos obtener el valor de una promesa, usamos `await`:

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

Pero sólo podemos usar `await` dentro de funciones marcadas con `async`:

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

Esto convierte el valor devuelto por la función en una promesa.

Promesas en peticiones HTTP

Usaremos promesas con la función `fetch()`, que realiza peticiones HTTP.

- `fetch(url)` devuelve una promesa con un objeto `Response`.
- Si queremos el contenido JSON, debemos llamar a `response.json()`, que también devuelve otra promesa.

Ejemplo de función asíncrona

```
async function trae_resultado(numero) {
  let base = "http://localhost:3000/";
  let recurso = "api/dobla/";
  let url = base + recurso + numero;

  let respuesta = await fetch(url);
  let datos = await respuesta.json();
  return datos;
}
```

Cadenas de async/await

- Si una función usa `await`, debe ser `async`.
- Si una función llama a otra función `async`, deberá usar `await`.
- Esto continúa hacia arriba en la jerarquía de llamadas.

En otras palabras: **si abajo hay un await, arriba habrá otro await, y otro, y otro...**

El servicio REST

Nuestro servidor tendrá:

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

Ficheros necesarios

Necesitamos:

- **dobla_client.html**: página web.
- **dobla_client.js**: código JavaScript del cliente.
- **dobla_server.js**: servidor Express que sirve la API y los ficheros.

dobla_client.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Cliente 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>
```

dobla_client.js (1/2)

```
"use strict";

async function trae_resultado(numero) {
  let base = "http://localhost:3000/";
  let recurso = "api/dobla/";
  let url = base + recurso + numero;

  let respuesta = await fetch(url);
  let datos = await respuesta.json();
  return datos;
}
```

dobla_client.js (2/2)

```
async function manej_boton01() {
  let numero = Math.floor(Math.random() * 100);

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

boton01.addEventListener("click", manej_boton01);
```

dobla_server.js

```
const express = require('express');
const path = require('path');
const app = express();

const puerto = 3000;
app.use(express.json());

const dir_raiz = path.join(process.env.HOME, "www/site01");
app.use(express.static(dir_raiz));

app.get('/api/dobla/:num', (req, res) => {
  let num = Number(req.params.num);
  let resultado = num * 2;
  res.json(resultado);
});

app.listen(puerto, () =>
  console.log(`Servidor en http://localhost:${puerto}`)
);
```

Comprobación del código de estado

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

if (respuesta.status === 200) {
  let datos = await respuesta.json();
  return datos;
} else {
  return "Error " + respuesta.status;
}
```

Si el código no es 200, no debemos intentar leer el JSON.