EDICIÓN #011

Qué es CORS y cómo funciona el Cross-Origin Resource Sharing

12 de Enero de 2026

Qué es CORS y cómo funciona el Cross-Origin Resource Sharing

Descubre qué es realmente CORS y por qué existe. Aprende a configurar los headers clave (Allow-Origin, Allow-Methods) y evita errores comunes con tokens y cookies.

10 December 2025

Hola,

Si todavía no te has pegado con CORS lo harás en algún momento de tu vida y si te has pegado sabes lo frustrante que puede ser las primeras veces que te topas con el típico error de: `Access to fetch at 'https://api.com' from origin 'http://localhost:3000' has been blocked by CORS policy…`

La verdad que es algo frustrante. CORS no es un bug, ni una configuración caprichosa para complicarte la vida. Es un mecanismo fundamental de **seguridad de la web** que, una vez entiendes cómo funciona por dentro deja de ser un dolor de cabeza.

Quería hacer este post porque ya son varias las veces que me he pegado con CORS y siempre caía en lo mismo a si que voy a tratar de explicarte todo lo que he aprendido del tema para que a ti no te pase.

Además, he desarrollado una herramienta para que entiendas CORS de forma visual que te explico al final. Vamos allá.

CORS Explicado: Qué es y por qué es fundamental para la Seguridad Web

Para entender CORS (*Cross-Origin Resource Sharing*), primero tienes que entender qué es la **Same-Origin Policy**.

Por defecto, los navegadores son muy paranoicos. Una web que esté alojada en el **punto A** no puede leer datos de un **origen B** salvo que B le dé permiso explícito para hacerlo.

Esto es lo que impide que si entras en una web maliciosa, su código JavaScript pueda hacer una petición a tu banco y leer tu saldo aprovechando que tienes la sesión iniciada.

Aquí es donde entra el **concepto de Origen**. Para el navegador, **dos URLs tienen el mismo origen solo si comparten exactamente tres cosas**:

- **Protocolo** (Scheme): http o https.
- **Dominio**: `https://www.google.com` y `https://api.google.com` (son distintos).
- **Puerto**: 80, 443, 3000…

Si cambias cualquiera de esos tres, el navegador lo entiende como un **origen distinto**.

Por tanto, `http://localhost:3000` y `http://localhost:8000` son orígenes diferentes. Si intentas conectarlos, el navegador bloquea la respuesta por seguridad… a menos que uses **CORS**.

¿Entonces qué pinta CORS en todo esto?

CORS es el mecanismo que permite que un servidor le diga explícitamente al navegador: *esta web de otro origen tiene permiso para acceder a mis datos*.

Cuando el navegador ve una cabecera como Access-Control-Allow-Origin: `https://lvrpiz.com`, interpreta que el servidor concede acceso y, por tanto, desbloquea la respuesta.

Si esa **cabecera no está presente** (o no coincide con el origen que hace la petición) el navegador protege al usuario y **bloquea el contenido**.

En otras palabras, **CORS es una capa de seguridad del navegador** que evita filtraciones de información entre sitios. Entonces tú, como desarrollador, solo necesitas configurarlo cuando quieres permitir que un frontend se comunique con un backend que vive en otro origen. Cuando está bien configurado, **CORS actúa como una whitelist** controlada por el servidor que especifica quién puede leer qué.

CORS No es lo que Crees: Lo que CORS NO hace

Antes de seguir, quiero aclarar un par de cosas porque veo mucha confusión al respecto:

- CORS no protege tu API frente a atacantes. Un atacante no depende del navegador, utiliza scripts o herramientas como curl, Python o Postman. **CORS no impide esas peticiones**.
- **No actúa como un firewall**. La petición llega al servidor y este la procesa con normalidad. El bloqueo ocurre después, cuando el navegador decide no mostrar la respuesta por falta de permisos.
- **No reemplaza la autenticación**. Dar permiso CORS a un origen solo permite que ese sitio lea la respuesta. El servidor sigue necesitando verificar si el usuario está autorizado a acceder a los datos.

Tipos de Peticiones CORS: Petición Simple vs Preflight

No todas las peticiones se tratan igual. El navegador distingue **dos categorías** y el comportamiento de CORS varía por completo según cuál estés usando.

1. **Peticiones simples**

Son las más básicas. Utilizan métodos GET, HEAD o POST y únicamente headers permitidos, como Content-Type con valores tradicionales tipo application/x-www-form-urlencoded.

En este caso **el navegador envía la petición directamente**. El servidor responde y, si falta el header CORS apropiado, el navegador **bloquea el acceso** a la respuesta.

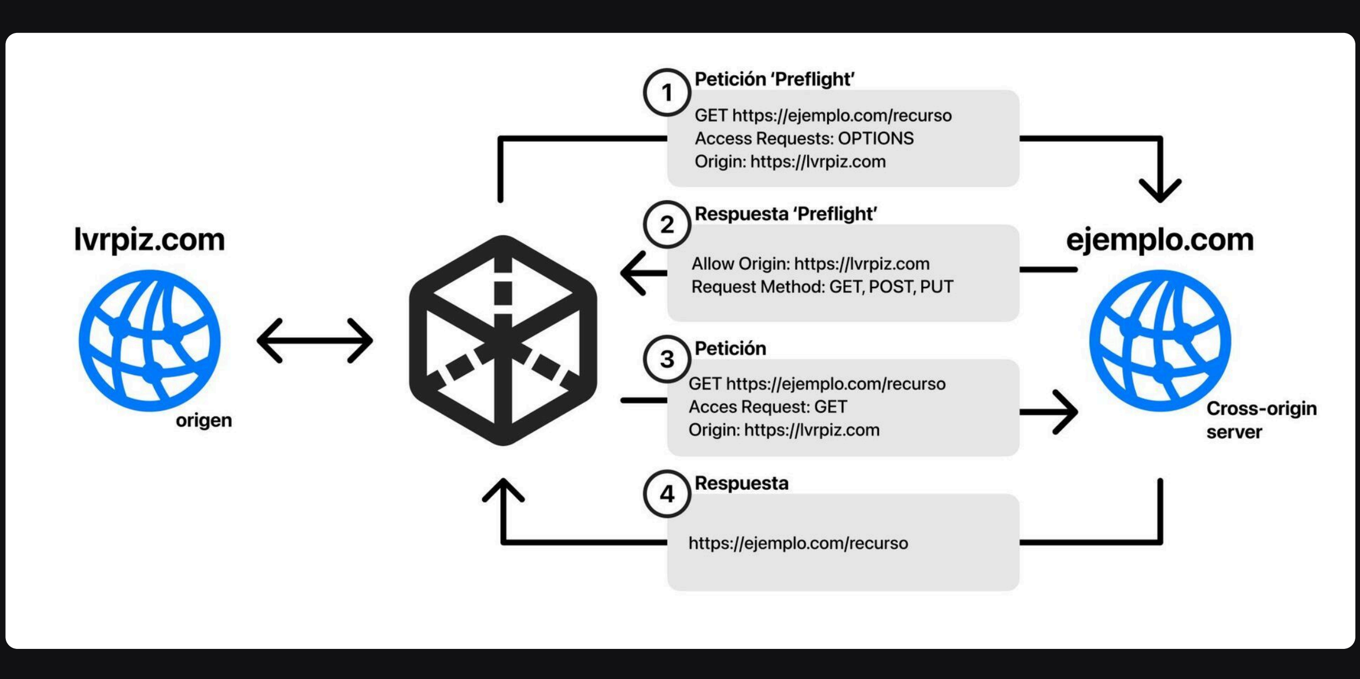
2. **Peticiones Preflight (o no simples)**

Aquí es donde entra en juego el mecanismo de verificación previa. Si tu petición utiliza métodos como PUT, DELETE, PATCH, envía JSON con application/json o incluye headers personalizados como Authorization, **el navegador necesita confirmar** primero que el **servidor acepta ese tipo de interacción**. ¿Cómo lo comprueba?

1. Primero se envía una petición automática de tipo OPTIONS. Es un paso previo que actúa como una consulta rápida como:

   "Oye `servidor`, soy `localhost:3000`, ¿puedo enviarte un PUT con un header Authorization?"

2. El servidor responde en base a su configuración devolviendo un estado 204 o similar junto con los headers CORS necesarios.
3. Por último el navegador habiendo validado eso envía la petición real.



Funcionamiento petición preflight CORS

Si esta verificación previa no obtiene el visto bueno, aparece el error de CORS y la **petición principal nunca llega a enviarse**.

Tipos de Headers CORS: Cómo configurar Access-Control-Allow-Origin para evitar errores en tu API

CORS funciona principalmente mediante **headers HTTP** que el servidor devuelve para indicarle al navegador **qué está permitido y qué no**. Configurarlos correctamente es clave para que tu frontend pueda comunicarse con tu API sin problemas. Los headers más importantes son:

- **Access-Control-Allow-Origin**: Especifica **qué dominios pueden acceder** a tu API. Puede ser un dominio concreto como `https://miweb.com` o un comodín * para permitir todos los orígenes.
- **Access-Control-Allow-Methods**: Lista los **métodos HTTP permitidos**, como *GET, POST, PUT, DELETE o OPTIONS*. Esto define qué tipo de acciones puede realizar el frontend.
- **Access-Control-Allow-Headers**: Necesario cuando envías tokens o headers personalizados. Debes listar explícitamente **headers** como *Authorization* o *X-API-Key* para que el navegador no bloquee la petición.
- **Access-Control-Allow-Credentials**: Si se establece como true, permite que el navegador **envíe cookies** o credenciales de sesión con la petición.
- **Access-Control-Max-Age**: Indica cuánto **tiempo (en segundos)** el navegador puede cachear la respuesta del preflight (OPTIONS). Así evitamos que se repita la petición de verificación con tanta frecuencia y mejorando el rendimiento.

Configurar estos headers correctamente asegura que tu aplicación funcione sin errores CORS y mantiene la seguridad que proporciona la política de mismo origen.

Solución a Fallos típicos de CORS: Preflight, Tokens y Allow-Origin Correcto

Ya puedes saberte toda la teoría de CORS que luego en la práctica es otro rollo. Aquí te enseño tres casos reales donde se suele fallar y cómo solucionarlo.

Caso 1: Desarrollo local con diferentes puertos

Ponte que tienes un frontend en React corriendo en `localhost:3000` y un backend en Express, Django o Go en `localhost:8000`.

Aquí tenemos un problema. Aunque el dominio y protocolo sean iguales, los **puertos son distintos**, lo que hace que el navegador considere orígenes diferentes, bloqueando la petición.

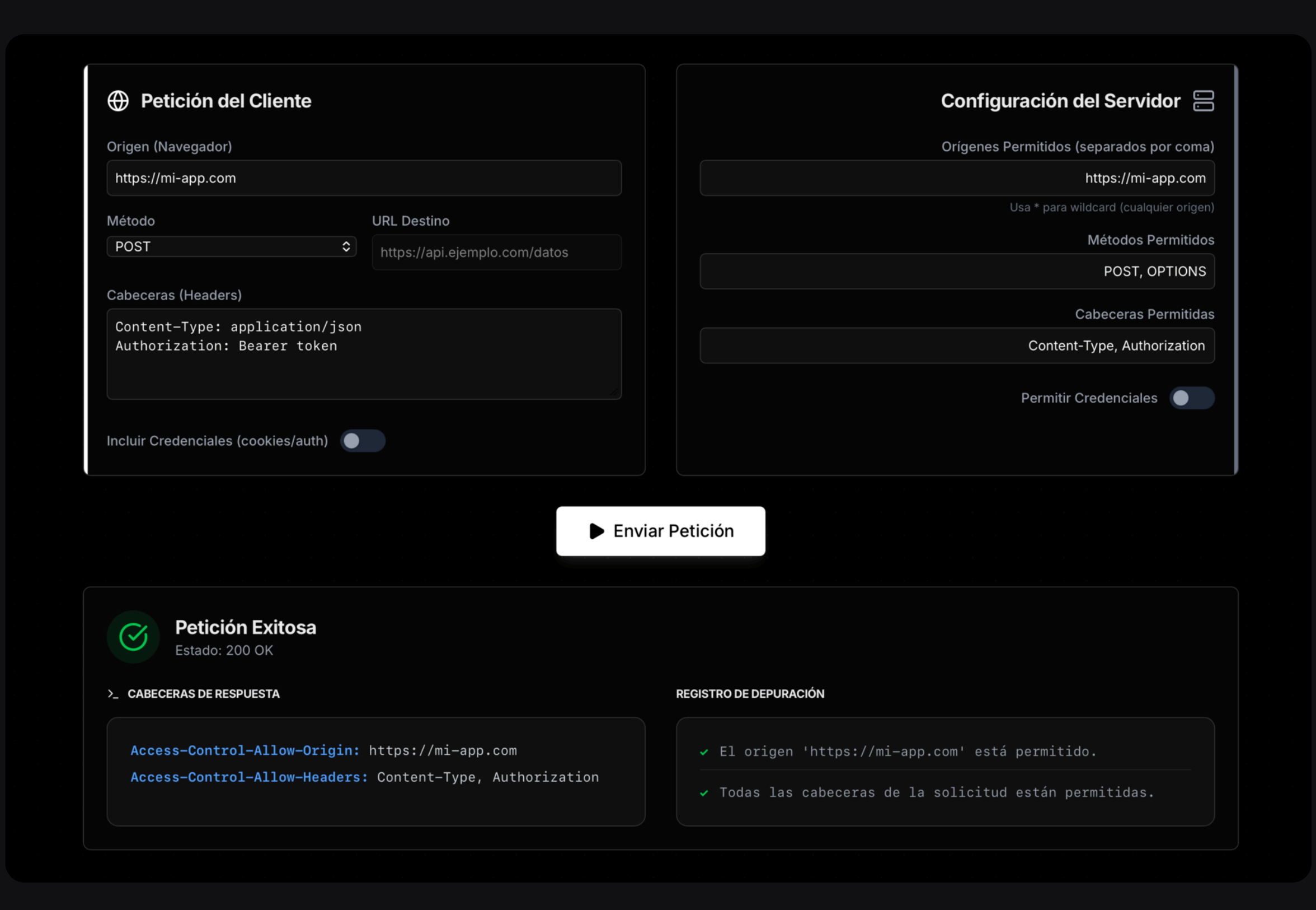
Solución: durante desarrollo, lo más sencillo es configurar el backend para devolver Access-Control-Allow-Origin: *. Esto permite que cualquier origen acceda a la API sin bloquearse.

OJO cuidado: **usar * en producción no es seguro**, porque permitiría que cualquier web acceda a tu API. En entornos reales, siempre debes especificar explícitamente los orígenes que quieres autorizar. Que no se te pase…

Caso 2: Peticiones JSON con token (Preflight)

Si haces un POST enviando JSON y un header de Authorization, el navegador ejecuta un preflight enviando una **petición OPTIONS antes del POST real**.

El error más común: el backend maneja el POST correctamente, pero **no responde al método OPTIONS en esa ruta**. Como resultado, el navegador recibe un 404 en el preflight y bloquea la petición.



Origen (Navegador): https://mi-app.com

Método: POST

URL Destino: https://api.ejemplo.com/datos

Cabeceras (Headers):

```
Content-Type: application/json
Authorization: Bearer token
```

Incluir Credenciales (cookies/auth) ○

Orígenes Permitidos (separados por coma): https://mi-app.com

Usa * para wildcard (cualquier origen)

Métodos Permitidos: POST, OPTIONS

Cabeceras Permitidas: Content-Type, Authorization

Permitir Credenciales ○

Petición Exitosa

Estado: 200 OK

CABECERAS DE RESPUESTA

```
Access-Control-Allow-Origin: https://mi-app.com
Access-Control-Allow-Headers: Content-Type, Authorization
```

REGISTRO DE DEPURACIÓN

```
✓ El origen 'https://mi-app.com' está permitido.
✓ Todas las cabeceras de la solicitud están permitidas.
```

Simulación de petición JSON con Token

Solución: asegúrate de que tu backend acepte y responda correctamente a las solicitudes OPTIONS para todas las rutas que requieran CORS.

Caso 3: Cookies y credenciales

Cuando quieres que la petición incluya cookies de sesión o autenticación, existen **dos reglas esenciales** que fallan en la mayoría de los casos:

- En el **frontend**, debes añadir credentials: 'include' (si usas fetch) o withCredentials: true (si usas axios).
- En el **backend**, no puedes usar * en Access-Control-Allow-Origin. Debes devolver explícitamente el origen que realiza la petición, por ejemplo http://localhost:3000. Si usas **wildcard** con Access-Control-Allow-Credentials: true, el **navegador bloqueará la petición** por motivos de seguridad.

Si sigues estas reglas vas a cubrir casi todos los problemas comunes de CORS que puedas tener en desarrollo y producción y evitarte errores difíciles de diagnosticar.

Errores Comunes de CORS en la Nube y Cómo Evitarlos: AWS API Gateway y Lambda

Seguramente muchos problemas que te encuentres de CORS no van a venir de tu código, sino de la **infraestructura** que sirve tu API o tus recursos.

Si usas AWS API Gateway o Lambda, la gestión de CORS normalmente se hace a nivel del **gateway**, no dentro de la función. Debes **habilitar CORS** en la consola de AWS para que el Gateway responda automáticamente a las solicitudes OPTIONS, que forman parte del preflight. Esto evita errores cuando tu frontend intenta hacer peticiones con métodos distintos a GET o POST simples.

Lo mismo aplica a **buckets S3** o servicios como **Cloudflare**. Si sirves recursos estáticos como fuentes, imágenes o scripts desde un CDN, debes **configurar correctamente las reglas CORS en el panel del proveedor**. Te dejo dos guías excelentes para configurar CORS en Cloudflare y en Amazon S3.

¿Quieres entender CORS de verdad? CORS Visualized

Mientras escribía este post pensé que era buena idea crear una herramienta para que cualquiera pueda experimentar cómo funciona sin romper su proyecto. CORS Visualized es un simulador interactivo donde puedes cambiar el origen, los métodos, las cabeceras y las credenciales de una petición, y ver al instante cómo reaccionan el servidor y el navegador.

Si alguna vez has querido entender CORS al completo te recomiendo echarle un vistazo, probar cosas, romperlas y si te ves confiado prueba el modo desafío que he metido.

La próxima vez que te pegues con CORS recuerda que no es algo que se pueda arreglar desde el **frontend**. No puedes saltártelo desde tu código en React, Vue u otro framework. Se trata de una **negociación entre navegador y servidor**.

El navegador actúa como agente del usuario, protegiendo sus datos, mientras que tu **servidor es quien decide en qué orígenes confía** y qué respuestas permite.

La próxima vez que veas las advertencias en rojo abre la pestaña Network en los DevTools, busca la petición OPTIONS y revisa los headers: ahí encontrarás siempre la clave de la solución.

Por último, para profundizar y entender bien cómo funciona, te recomiendo leer esta guía de MDN: CORS – MDN Web Docs.

Hasta el próximo miércoles,

Álvaro