2025 Volume: 01 Issue: 07 www.ijrpa.com ISSN 2456-9995 Review Article

International Journal Research Publication Analysis

Page: 01-29

AN IN-DEPTH ANALYSIS OF CROSS-ORIGIN RESOURCE SHARING (CORS) IN MODERN WEB APPLICATIONS

***Rinki Kumari, Dr. Vishal Shrivastava, Dr. Akhil Pandey**

Computer Science & Engineering, Arya College of Engineering & I.T. Jaipur, India.

Article Received: 29 October 2025
Article Revised: 18 November 2025
Published on: 09 December 2025

***Corresponding Author: Rinki Kumari**
Computer Science & Engineering, Arya College of Engineering & I.T. Jaipur, India. DOI: https://doi-doi.org/101555/ijrpa.7293

The Cornerstone of Web Security: The Same-Origin Policy (SOP)

The modern web is an intricate tapestry of documents, scripts, and resources loaded from countless different servers. A single web page can simultaneously display images from a content delivery network, run analytics scripts from a marketing service, and fetch data from a backend API. This interconnectedness, while powerful, presents a significant security challenge. Without a foundational security model, a malicious script from one website could potentially access and exfiltrate sensitive user data from another, completely compromising user privacy and security. The mechanism that prevents this chaotic scenario is a browser-enforced security feature known as the Same-Origin Policy (SOP). Understanding the SOP is not merely a prelude to understanding Cross-Origin Resource Sharing (CORS); it is the fundamental context that necessitates its very existence. CORS was designed not to replace the SOP, but to provide a standardized, controlled mechanism for relaxing its strictures when legitimate cross-domain communication is required.

Defining an "Origin": The Scheme, Host, and Port Tuple

At the heart of the Same-Origin Policy is the concept of an "origin." An origin is not simply the domain name of a website; it is a precisely defined construct determined by a combination of three distinct parts of a URL: the scheme (protocol), the host (domain name), and the port number. This combination is often referred to as the "scheme/host/port tuple". Two URLs are considered to have the same origin if, and only if, all three of these components are identical. For example, consider the URL https://www.example.com/index.html:

- **Scheme:** https

- **Host:** www.example.com
- **Port:** 443 (implied default for HTTPS)

URL 1	URL 2	Same Origin?	Reason for Difference
http://store.company.com/dir/page.html	http://store.company.com/dir2/other.html	Yes	Only the path differs.
http://store.company.com/dir/page.html	https://store.company.com/dir/page.html	No	Different scheme (protocol).
http://store.company.com/dir/page.html	http://store.company.com:81/dir/page.html	No	Different port.
http://store.company.com/dir/page.html	http://news.company.com/dir/page.html	No	Different host.

Any change to one of these components results in a different origin. This strict definition means that http://www.example.com is a different origin from https://www.example.com because the scheme differs. Likewise, https://www.example.com and https://api.example.com are different origins because their hosts are not the same, even though they share a parent domain. Finally, https://www.example.com and https://www.example.com:8080 are different origins due to the explicit port number in the second URL. This granular definition forms the bedrock of the browser's security model, dictating the boundaries for data interaction.

Historical Context and Rationale: Protecting User Data in a Multi-Tab World

The Same-Origin Policy was introduced by Netscape Navigator 2.02 in 1995, shortly after the integration of JavaScript into the browser. The advent of JavaScript transformed static web documents into dynamic applications by enabling programmatic access to the Document Object Model (DOM). This newfound power, however, introduced a new class of security risks. Without a clear boundary, a script loaded from one origin could freely manipulate the DOM of a page loaded from another origin.

The primary rationale behind the SOP is to isolate potentially malicious documents and scripts, thereby reducing the available attack vectors. It operates on a "default deny" security posture, a fundamental principle of information security where access is forbidden unless explicitly granted. The SOP assumes that any cross-origin interaction is potentially hostile and blocks it by default. This is critical in a multi-tab browsing environment. Imagine a user is logged into their online banking portal at https://mybank.com in one tab and simultaneously visits a malicious website, https://evil-site.com, in another. Without the SOP, a script running on evil-site.com could make a request to https://mybank.com/account-details. Because the user is authenticated with mybank.com, their browser would automatically include their session cookie with the request. The mybank.com server would see a valid, authenticated request and return the user's account information. The SOP prevents this attack by blocking the script on evil-site.com from being able to read the response from mybank.com, thus protecting the confidentiality of the user's data.

SOP in Action: What is Permitted vs. What is Blocked

The Same-Origin Policy does not block all cross-origin interactions. Its rules are nuanced, generally distinguishing between embedding resources (or writing data) and reading data. **Permitted Cross-Origin Actions:** The SOP historically permitted certain types of cross-origin interactions, largely because they were considered "write-only" or because they predated the sophisticated scripting that could exploit them for data theft.

- **Embedding Resources:** Web pages are fundamentally designed to be compositions of resources. Therefore, embedding cross-origin resources using HTML tags is generally allowed. This includes images via <img>, scripts via <script>, stylesheets via <link>, and multimedia via <video> and <audio>. While the resource is displayed or executed, the embedding page's script cannot typically inspect its contents.
- **Form Submissions:** An HTML <form> can have its action attribute point to a cross-origin URL. This allows a page from one origin to submit, or "write," data to another. This is a foundational mechanism of the web, but it is also the vector for Cross-Site Request Forgery (CSRF) attacks, as the SOP does not prevent these writes.
- **iframes:** A page can embed a cross-origin iframe. However, the SOP severely restricts the ability of the parent page's script to access the content and DOM of the embedded page, and vice-versa.

Blocked Cross-Origin Actions: The primary restriction enforced by the SOP is on programmatic read access to cross-origin resources.

- **Scripted HTTP Requests:** The most significant restriction is on requests made via JavaScript using APIs like fetch() or XMLHttpRequest. A script on one origin is blocked from making such a request to another origin and reading the response. This is the core protection against the data theft scenario described previously.

- **DOM Access:** A script cannot access the DOM of a cross-origin document, such as one loaded in an iframe. This prevents a malicious page from reading sensitive information or manipulating the user interface of a trusted site embedded within it.
- **Browser Storage:** Access to data stored in the browser, such as LocalStorage, SessionStorage, and IndexedDB, is strictly partitioned by origin. JavaScript from one origin cannot read from or write to the storage belonging to another origin.

Inherent Limitations and the Rise of Web APIs

While indispensable for security, the strictness of the SOP became a significant obstacle to the evolution of the web. The rise of Single-Page Applications (SPAs) and microservice architectures created a paradigm where web applications are often composed of a frontend served from one origin (e.g., https://app.example.com) that needs to communicate with one or more backend APIs served from different origins (e.g., https://api.example.com, https://auth.example.com). Furthermore, the proliferation of third-party APIs for services like payment processing, social media integration, or data visualization meant that applications increasingly needed to make legitimate, secure cross-origin requests.

This created a fundamental tension between the web's foundational security model and the architectural demands of modern applications. The SOP, in its default state, blocked these legitimate interactions. Developers initially resorted to workarounds like JSONP, which carried significant security risks. It became clear that a standardized, secure mechanism was needed to allow servers to explicitly and granularly grant permission for cross-origin reads. This need was the direct catalyst for the development and standardization of Cross-Origin Resource Sharing (CORS).

The CORS Protocol: A Detailed Technical Examination

Cross-Origin Resource Sharing (CORS) is a W3C standard that extends the Same-Origin Policy with a set of HTTP headers. It provides a mechanism for a server to explicitly permit a web browser to make cross-origin requests from a specified set of origins. It is not a new security policy but rather a controlled protocol for relaxing the existing one. The entire CORS mechanism is a negotiation between the client (browser) and the server, where the browser initiates the request with information about its origin, and the server responds with a policy that the browser then enforces. This server-driven, browser-enforced model allows for the creation of rich, integrated web applications without compromising the fundamental security principles of the web.

The Anatomy of a CORS Request: Simple vs. Preflighted Requests

The CORS standard categorizes cross-origin requests into two main types: "simple requests" and "preflighted requests." This distinction is critical because it determines whether the browser can send the request directly or must first send a preliminary "preflight" request to ask for the server's permission.

Simple Requests

A simple request is a CORS request that is sent directly to the server without a preceding preflight check. The browser makes the actual request (e.g., a GET request) and includes the Origin header. It then inspects the response headers from the server. If the server's response includes the appropriate CORS headers (like Access-Control-Allow-Origin) that permit the request, the browser allows the client-side JavaScript to access the response. If not, the request fails from the perspective of the JavaScript code.

For a request to be classified as "simple," it must meet a strict set of conditions. This is because these types of requests are similar in nature to what was possible before CORS (e.g., via HTML form submissions), and thus were considered less likely to introduce new security risks to legacy servers. The conditions are as follows :

- **HTTP Method:** The request must use one of the following methods:
  - GET
  - HEAD
  - POST
- **HTTP Headers:** Apart from headers automatically set by the user agent (like Host or User-Agent), the only headers that can be manually set are the CORS-safelisted request-headers:
  - Accept
  - Accept-Language
  - Content-Language
  - Content-Type
  - Range
- **Content-Type Header Value:** If the Content-Type header is present, its value must be one of the following media types:
  - application/x-www-form-urlencoded
  - multipart/form-data

- text/plain

- **Other Constraints:** No event listeners can be registered on the XMLHttpRequest.upload object, and no ReadableStream object can be used in the request.
  Any request that violates even one of these conditions is not a simple request and must be preflighted. A common example that triggers a preflight is a POST request with a Content-Type of application/json, which is the standard for modern REST APIs.

Table 2: Conditions for a CORS "Simple Request"

Criterion	**Allowed Values / Conditions**
Method	GET, HEAD, POST
Headers	Only CORS-safelisted headers (Accept, Accept-Language, Content-Language, Content-Type, Range) may be manually set.
Content-Type Value	application/x-www-form-urlencoded, multipart/form-data, text/plain
XMLHttpRequest	No event listeners on the upload property.
Request Body	No ReadableStream object used.

Preflighted Requests

For any request that is not simple, the browser must first send a preliminary "preflight" request to the server to ensure the server understands and approves of the actual request that is to follow. This preflight mechanism is a brilliant piece of backward-compatible design. Before CORS, a server at api.example.com would never expect to receive a cross-origin DELETE request from a browser script. Its logic could have been built on the assumption that such requests were impossible under the SOP. If a modern browser simply sent the DELETE request, this legacy, non-CORS-aware server might process it, leading to unintended side effects. The preflight request solves this by acting as an "opt-in" signal. A non-CORS-aware server will not know how to respond correctly to the preflight, causing the browser to abort the actual, potentially harmful DELETE request, thus protecting the old server.

The preflight request itself is an HTTP OPTIONS request sent automatically by the browser to the same URL as the actual request. This OPTIONS request includes special headers that describe the intended actual request :

- Access-Control-Request-Method: Specifies the HTTP method that the actual request will use (e.g., PUT, DELETE).

- Access-Control-Request-Headers: Specifies any non-simple headers that the actual request will include (e.g., Content-Type, Authorization).

The server, upon receiving the preflight OPTIONS request, inspects these headers and determines if it is willing to accept such a request. It then responds with its own set of CORS headers. If the server's response indicates that the method and headers are permitted for that origin, the browser then proceeds to make the actual HTTP request. If the preflight is denied, the actual request is never sent, and an error is reported in the browser's console.

To improve performance, the response to a preflight request can be cached by the browser. The server can include the Access-Control-Max-Age header in its preflight response, specifying the number of seconds the browser can cache the permissions. This avoids the need to send a new preflight request for every subsequent non-simple request to the same resource.

The Language of CORS: A Comprehensive Review of HTTP Headers

The entire CORS protocol is orchestrated through a set of standardized HTTP headers exchanged between the browser and the server. A thorough understanding of these headers is essential for both implementing and debugging CORS.

Table 3: Comprehensive CORS HTTP Header Reference.

Header	Type	Purpose	Example Value(s)	Context of Use
Origin	Request	Indicates the origin (scheme, host, port) of the script Initiating the request.	https://www.example.com	All CORS requests.

Access-Control-Request-Method	Request	Sent in a preflight request to inform the server of the HTTP method to be used in the actual request.	PUT	Preflight (OPTIONS) requests only.
Access-Control-Request-Headers	Request	Sent in a preflight request to inform the server of the non-simple HTTP headers to be used in the actual request.	Content-Type, Authorization	Preflight (OPTIONS) requests only.
Access-Control-Allow-Origin	Response	Specifies the origin that is permitted to access the resource. The most critical CORS response header.	https://www.example.com or *	All CORS responses, including preflight.
Access-Control-Allow-Methods	Response	Specifies the method(s) allowed when accessing the resource.	GET, POST, PUT, DELETE	Preflight (OPTIONS) responses only.
Access-Control-Allow-Headers	Response	Specifies the header(s) allowed in the actual request.	Content-Type, Authorization	Preflight (OPTIONS) responses only.
Access-Control-Allow-Credentials	Response	Indicates whether the response to a request with credentials can be exposed to the requesting script.	true	All CORS responses where credentials are sent.

Access-Control-Max-Age	Response	Indicates how long the results of a preflight request can be cached in seconds.	86400	Preflight (OPTIONS) responses only.
Access-Control-Expose-Headers	Response	Whitelists headers in the response that JavaScript in browsers is allowed to access.	Content-Length, X-My-Custom-Header	Responses to actual requests (not preflight).

Handling Authenticated Requests: withCredentials and Access-Control-Allow-Credentials

By default, for security reasons, browsers do not include credentials such as cookies, HTTP authentication headers, or TLS client certificates in cross-origin requests. This is a critical safeguard against CSRF-style attacks where a malicious site could trigger an authenticated action on another domain.

To enable the sending of credentials on a cross-origin request, a two-part handshake is required, involving explicit consent from both the client-side script and the server:

1. **Client-Side Opt-In:** The client-side code must explicitly signal its intent to include credentials. This is done by setting the withCredentials property to true on an XMLHttpRequest object or by using the credentials: 'include' option in a fetch() request.
2. **Server-Side Permission:** The server must explicitly permit the request with credentials by including the Access-Control-Allow-Credentials: true header in its response.
3. Both conditions must be met. If the client sends withCredentials: true but the server does not respond with Access-Control-Allow-Credentials: true, the browser will reject the response and the request will fail.

There is a crucial security constraint tied to this mechanism: when a server responds with Access-Control-Allow-Credentials: true, the Access-Control-Allow-Origin header **must** specify a single, explicit origin. It cannot be the wildcard *. This prevents a scenario where a server inadvertently exposes credential-protected resources to any website on the internet. If a browser receives a response with both Access-Control-Allow-Credentials: true and Access-

Control-Allow-Origin: *, it will block the response as a security violation.

The Security Landscape: CORS Misconfigurations and Vulnerabilities

While CORS is a mechanism designed to enable functionality, its security is entirely dependent on its correct implementation. Misconfigurations are common and can lead to serious vulnerabilities that undermine the very protections the Same-Origin Policy was created to provide. The security of a CORS policy relies on the principle of whitelisting—explicitly defining what is allowed—rather than blacklisting. Most vulnerabilities arise from failures in implementing a sufficiently strict and accurate whitelist.

Common Pitfalls in Access-Control-Allow-Origin Configuration

The Access-Control-Allow-Origin header is the linchpin of any CORS policy, and it is the most frequent source of security flaws.

The Dangers of the Wildcard (*)

Setting Access-Control-Allow-Origin: * is the most permissive configuration. It signals to browsers that any origin is allowed to make a request and read the response. While this may be acceptable for truly public, unauthenticated APIs (e.g., a public font library), it is extremely dangerous for any API that handles sensitive or user-specific data. If an API that returns private user information is configured with a wildcard origin, any malicious website can use JavaScript to make a request on behalf of a logged-in user, read the sensitive data from the response, and exfiltrate it.

As previously noted, the CORS specification provides a critical built-in defense against the most egregious version of this flaw: it is invalid to use the wildcard origin (*) in conjunction with Access-Control-Allow-Credentials: true. A browser will reject such a response, preventing a malicious site from easily stealing data from an authenticated session.

Insecure Origin Reflection

A common but highly insecure practice is for a server to dynamically generate the Access-Control-Allow-Origin header by simply reading the Origin header from the incoming request and reflecting its value back in the response. Developers often implement this with the mistaken belief that it is a flexible way to allow all origins.

This configuration completely nullifies the Same-Origin Policy. An attacker can host a malicious script on https://evil-site.com. When this script makes a request to the vulnerable server, the browser will send the header Origin: https://evil-site.com. The server, seeing this, will dutifully respond with Access-Control-Allow-Origin: https://evil-site.com, thereby granting the malicious script full permission to read the response.

Whitelist Parsing Flaws

Even when developers attempt to implement a proper whitelist of allowed origins, subtle errors in the validation logic can be exploited. These flaws often arise from using overly simplistic string matching instead of precise validation.

- **Suffix Matching:** A rule that checks if the request Origin ends with .trusted.com can be bypassed by an attacker registering the domain malicious-trusted.com.
- **Prefix Matching:** A rule that checks if the Origin starts with trusted.com can be bypassed by registering trusted.com.evil.net.
- **Substring Matching:** A rule that simply checks if trusted.com is present anywhere in the Origin string is similarly vulnerable.

To be secure, origin validation must use exact string matching against a list of known-good origins or a carefully crafted regular expression that correctly anchors the domain name (e.g., ^https://(.*\.)?trusted\.com$).

The null Origin Vulnerability

In certain specific circumstances, such as requests originating from a file:// URL or from a sandboxed iframe, the browser will send the header Origin: null. Developers, often in an attempt to facilitate local testing, might add null to their server's origin whitelist.

This creates a significant vulnerability. An attacker can craft a malicious web page that uses a sandboxed iframe to make a request to the target API. The browser will send Origin: null, which the server will then accept because it is on the whitelist. This grants the attacker's script access to the API response, which it can then exfiltrate. For this reason, Access-Control-Allow-Origin:
Null should be avoided in production environments.

Exploiting Trust: How XSS in a Whitelisted Origin Compromises Security

A correctly configured CORS policy establishes a trust relationship between two origins. For instance, if an API server at https://api.example.com sets Access-Control-Allow-Origin: https://app.example.com, it is explicitly trusting that the app.example.com origin is not malicious. This trust can be exploited if the whitelisted origin is itself vulnerable. If an attacker discovers a Cross-Site Scripting (XSS) vulnerability on https://app.example.com, they can inject malicious JavaScript into that page. When a victim visits the compromised page on app.example.com, the attacker's script executes with the full privileges of that origin. The script can then make a CORS request to https://api.example.com. From the API server's perspective, this is a perfectly valid request, as it originates from the trusted https://app.example.com. The API server will grant access, and the attacker's script can then read the sensitive response and send it to a server under the attacker's control. This demonstrates that the security of a CORS-protected resource is dependent on the security of all the origins it chooses to trust.

CORS and Cross-Site Request Forgery (CSRF): A Clarification

A pervasive and dangerous misconception among developers is that CORS provides protection against Cross-Site Request Forgery (CSRF) attacks. **CORS is not a defense against CSRF**.

A CSRF attack works by tricking an authenticated user's browser into submitting an unintended, state-changing request to a trusted site. For example, an attacker could embed a hidden form on a malicious page that, when submitted, transfers money from the victim's bank account. The SOP does not prevent cross-origin writes, such as form submissions or simple POST requests. The browser will send the forged request, and because the user is authenticated, it will automatically include their session cookies, making the request appear legitimate to the server. The primary role of the SOP and CORS in this context is to prevent the attacker's page from reading the response to that forged request. The state-changing action itself, however, may still succeed.

Worse, a poorly configured CORS policy can actually escalate the impact of a CSRF vulnerability. Consider an endpoint that is vulnerable to CSRF and also has an overly permissive CORS policy (e.g., reflecting the Origin header). An attacker could use fetch() from their malicious site to not only trigger the state-changing action but also to read the response. This could allow them to confirm the success of the attack or steal sensitive information (such as a CSRF token for a subsequent request or personal data) returned in the response body, something that would be impossible with a proper CORS policy in place.

Recommendations for a Secure CORS Policy

Based on common vulnerabilities and security best practices from organizations like OWASP, a secure CORS policy should adhere to the following principles:

- **Use a Strict Whitelist:** Always define a specific and explicit list of trusted origins. Avoid using Access-Control-Allow-Origin: * in production unless the API is intended to be fully public and unauthenticated.
- **Validate Dynamically:** If the list of allowed origins is dynamic, the server-side code must rigorously validate the incoming Origin header against this list before reflecting it. Do not blindly reflect the origin.
- **Be Precise:** Use exact string matching for origin validation. Avoid partial matching (e.g., startsWith, endsWith, contains) or poorly constructed regular expressions that can be bypassed.
- **Deny null Origin:** Do not include the null origin in your production whitelist.
- **Apply Least Privilege:** Only allow the HTTP methods and headers that are strictly necessary for the application's functionality in the Access-Control-Allow-Methods and Access-Control-Allow-Headers headers.
- **Handle Credentials with Caution:** Only use Access-Control-Allow-Credentials: true when absolutely necessary, and always pair it with a specific, non-wildcard origin.
- **Maintain Defense in Depth:** Remember that CORS is a browser-level security feature. It does not replace the need for robust server-side security measures, including proper authentication, authorization, input validation, and strong CSRF defenses (like synchronizer tokens).

CORS in Practice: Secure Server-Side Configuration

Implementing a secure CORS policy requires translating the theoretical principles of whitelisting and least privilege into concrete server configurations. The approach varies depending on the web server or application framework, but the goal remains the same: to establish a granular and restrictive policy that allows legitimate application traffic while blocking all else. Secure configuration is not a global, server-wide toggle but rather a deliberate, often per-endpoint, set of rules.

Implementing a Secure CORS Policy in Apache

On the Apache HTTP Server, CORS headers are managed using the mod_headers module, which is typically enabled by default. Configurations can be placed within a <VirtualHost>,

<Directory>, <Location> block, or in a .htaccess file.

A naive and insecure approach is to simply add the header for all requests: Header set Access-Control-Allow-Origin "*"

A more secure implementation involves conditionally setting the header only for trusted origins. This can be achieved by checking the incoming Origin header against a whitelist.

Secure Apache Configuration Example:

```
# Ensure mod_headers is enabled # a2enmod headers

<IfModule mod_headers.c>
# Define a regular expression for allowed origins SetEnvIf Origin
"^https?://(www\.)?(trusted-app\.com|another-trusted-domain\.org)$"
ACAO_ORIGIN=$0

# Set the ACAO header only if the Origin matched the whitelist Header set Access-Control-
Allow-Origin %{ACAO_ORIGIN}e
env=ACAO_ORIGIN
# Set Vary: Origin to handle caching correctly Header append Vary Origin

# Handle preflight OPTIONS requests RewriteEngine On
RewriteCond %{REQUEST_METHOD} OPTIONS
RewriteRule ^(.*)$ $1
</IfModule>
```

In this example, SetEnvIf is used with a regular expression to check if the Origin header matches one of the whitelisted domains. If it does, an environment variable ACAO_ORIGIN is set to the value of the matched origin. The Header set directive then uses this environment variable to dynamically but safely set the Access-Control-Allow-Origin header. The Vary: Origin header is appended to ensure correct behavior with caching proxies. Finally, a RewriteRule is used to gracefully handle preflight OPTIONS requests by returning a 204 No Content response without further processing.

Implementing a Secure CORS Policy in Nginx

Nginx offers a highly efficient and secure way to manage CORS policies using the map directive. This approach is generally preferred over using if statements inside a location block, as it is more performant and avoids common pitfalls associated with the if directive.

The map directive allows the creation of a variable whose value depends on other variables, in this case, the request method ($request_method) and the origin ($http_origin).

Secure Nginx Configuration Example:

This configuration should be placed in the http block of your nginx.conf.

```
# Map the request origin and method to a variable if it's on the whitelist
map "$request_method $http_origin" $allow_cors { default 0;
"~^OPTIONS https?://(www\.)?trusted-app\.com$" 1;
"~^(GET|POST) https?://(www\.)?trusted-app\.com$" 1;
"~^OPTIONS https?://(www\.)?another-domain\.org$" 1;
"~^(GET|POST) https?://(www\.)?another-domain\.org$" 1;
}

server {
#... other server configuration...

location /api/ {
# Only add headers if the request is from an allowed origin if ($allow_cors) {
add_header 'Access-Control-Allow-Origin' "$http_origin"
always;
add_header 'Access-Control-Allow-Methods' 'GET, POST,
OPTIONS' always;
add_header 'Access-Control-Allow-Headers' 'Content-Type,
Authorization' always;
add_header 'Access-Control-Allow-Credentials' 'true'
always;

}
add_header 'Vary' 'Origin' always;
# Handle preflight requests
if ($request_method = 'OPTIONS') {
# Only respond with 204 if the origin is allowed if ($allow_cors) {
return 204;
}
# Deny preflight from other origins return 403;
}
#... proxy_pass or other location logic...
}
}
```

This configuration first defines a map that sets the $allow_cors variable to 1 only if the request is an OPTIONS, GET, or POST from one of the whitelisted domains. Inside the location block, it checks this variable. If it's set, the appropriate CORS headers are added. The preflight OPTIONS request is handled specifically, returning a 204 No Content for allowed origins and terminating the request.

Implementing a Secure CORS Policy in Node.js with Express

For Node.js applications using the Express framework, the most common and recommended way to handle CORS is with the cors middleware package. While a simple app.use(cors()) will enable CORS for all origins, this is not secure for production environments. The package provides a rich configuration object to implement a strict policy.

The most secure approach is to use a dynamic origin function, which allows for programmatic validation of the requesting origin against an allowlist.

Secure Express.js Configuration Example:

```
const express = require('express'); const cors = require('cors'); const app = express();
// Whitelist of allowed origins
const allowedOrigins = ['https://trusted-app.com', 'https://another-trusted-domain.org'];
const corsOptions = {
origin: function (origin, callback) {
// Allow requests with no origin (like mobile apps or curl requests)
if (!origin) return callback(null, true);
if (allowedOrigins.indexOf(origin) === -1) {
```

```
const msg = 'The CORS policy for this site does not allow access from the specified Origin.';
return callback(new Error(msg), false);
}
return callback(null, true);
},
methods:, allowedHeaders:, credentials: true,
optionsSuccessStatus: 200 // For legacy browser support
};

// Enable pre-flight across-the-board app.options('*', cors(corsOptions));
// Apply CORS policy to a specific route app.get('/api/data', cors(corsOptions), (req, res) => {
res.json({ message: 'This is secure data.' });
});
app.listen(3000, () => {
console.log('Server running on port 3000 with secure CORS policy.');
});
```

In this example, a corsOptions object is defined. The origin property is a function that checks if the incoming origin is present in the allowedOrigins array. If it is not, it calls the callback with an error, which will cause the request to be rejected. This provides a robust and maintainable way to manage a whitelist of trusted clients. The policy is then applied selectively to specific routes, adhering to the principle of least privilege.

The Critical Role of the Vary: Origin Header in Caching Environments

When a server's CORS policy is dynamic—that is, when the Access-Control-Allow-Origin header's value can change based on the incoming request's Origin header—it creates a potential conflict with intermediate caches, such as Content Delivery Networks (CDNs) or reverse proxies.

Consider a scenario without the Vary header. A request arrives from a legitimate origin, https://good.com. The server responds with Access-Control-Allow-Origin: https://good.com, and the CDN caches this response. Moments later, a request for the same resource arrives from a different legitimate origin, https://another.com. The CDN, unaware that the response depends on the Origin header, serves the cached response intended for https://good.com. The browser of the user from https://another.com receives a response with Access-Control-Allow-

Origin: https://good.com, sees the mismatch, and blocks the request, causing the application to fail.

This can also lead to cache poisoning, where a response for a malicious origin could be cached and served to legitimate users.

The Vary HTTP response header is the solution to this problem. By including Vary: Origin in the response, the server signals to all caches that the response is not static and varies based on the value of the request's Origin header. This instructs the cache to use the Origin header as part of its cache key, storing a separate version of the response for each unique origin.
Therefore, for any dynamic CORS configuration, the Vary: Origin header is not merely a best practice but a mandatory component for ensuring both correctness and security.

CORS in Contemporary Architectures and Alternative Strategies

The modern web is increasingly built on distributed architectures. Single-Page Applications (SPAs) have decoupled the frontend from the backend, and microservice architectures have broken monolithic backends into smaller, independent services. In this landscape, cross-origin communication is not an edge case but the default mode of operation, making CORS an indispensable part of the web's infrastructure. However, CORS is not the only method for handling cross-origin data exchange; understanding its alternatives—both legacy and architectural—provides a clearer picture of its role and value. The choice between these methods often represents a trade-off between security, complexity, and control.

The Indispensable Role of CORS in SPAs and Microservices

SPAs, built with frameworks like React, Angular, or Vue.js, function by loading a single HTML shell and then dynamically fetching data and updating the UI using JavaScript APIs like fetch. In a typical deployment, the static assets of the SPA (HTML, CSS, JavaScript) are served from one origin (e.g., a CDN or a static web server at https://app.example.com), while the dynamic data is provided by backend APIs hosted on a different origin (e.g., https://api.example.com).

This architectural separation means that nearly every API call made by the SPA is a cross-origin request. Without a mechanism like CORS, these requests would be blocked by the browser's Same-Origin Policy, rendering the application non-functional. CORS provides the standardized, secure protocol that allows the API server at api.example.com to explicitly grant permission to the frontend at app.example.com to access its resources. This enables the clean separation of concerns between frontend and backend development, a cornerstone of modern web engineering.

Similarly, in a microservices architecture, a single user-facing application might need to aggregate data from multiple backend services, each running on its own domain or port. CORS facilitates this direct client-to-microservice communication, although in many complex systems, an API gateway is used to consolidate these services behind a single origin.

Comparative Analysis of Cross-Origin Communication Techniques

While CORS is the modern standard, it is important to understand it in the context of other techniques that have been used to solve the cross-origin problem.

JSON with Padding (JSONP): A Legacy "Hack"

Before CORS was widely adopted, developers used a clever but insecure workaround called JSONP (JSON with Padding) to retrieve data from different domains.

- **Mechanism:** JSONP exploits the fact that HTML <script> tags are not subject to the Same-Origin Policy for execution. A request is made by dynamically creating a <script> element and setting its src attribute to the target API endpoint. A special query parameter, typically named callback, is included in the URL (e.g., ?callback=myFunction). The server, instead of returning raw JSON, wraps the JSON data in a JavaScript function call using the name provided in the callback parameter (e.g., myFunction({"data": "value"})). When the script loads in the browser, the function is executed, passing the data to the client-side code.
- **Capabilities & Limitations:** JSONP is inherently limited to GET requests, as it relies on the <script src="..."> attribute. It also has very poor error handling capabilities; a failed request often results in a generic script error with no access to HTTP status codes.
- **Security Flaws:** JSONP is fundamentally insecure. By using it, a website is effectively allowing a third-party server to inject and execute arbitrary JavaScript code within its own origin. If the third-party server is compromised, an attacker can replace the benign data with a malicious script, leading to a full-blown Cross-Site Scripting (XSS) attack. This requires an absolute and often unwarranted level of trust in the remote server. For these reasons, JSONP has been rendered obsolete by CORS for all modern web applications.

Server-Side Proxies: A Functional Bypass

A server-side proxy is an architectural pattern used to circumvent browser-based cross-origin restrictions, especially when dealing with third-party APIs that do not support CORS.

- **Mechanism:** Instead of the client-side JavaScript making a direct cross-origin request to the target API, it makes a same-origin request to its own backend server. This backend server then acts as a proxy, making a server-to-server request to the target API. Since the Same-Origin Policy does not apply to server-to-server communication, this request succeeds. The proxy server receives the response from the API and then relays it back to the client.
- **Capabilities & Security:** This approach is highly capable, as it can support any HTTP method, header, or data format. From a security perspective, it has distinct pros and cons:
  - **Pros:** It is an excellent way to interact with APIs that require secret keys or tokens. The secret key can be stored securely on the proxy server and added to the outbound request, never exposing it to the client-side code. It also provides a crucial security benefit by preventing the user's browser from automatically attaching credentials (like cookies for the target domain) to the request, as the request from the browser only goes to the same-origin proxy server.
  - **Cons:** This pattern introduces additional infrastructure complexity and latency. The proxy server itself becomes a point of trust and a potential bottleneck. If configured as an "open proxy" (allowing requests to any arbitrary destination), it can be abused by attackers to launch attacks on other systems or to probe the proxy's internal network. The proxy has the ability to read and modify all request and response data that passes through it, which can be a security concern if the proxy itself is compromised.

The choice between these methods can be framed as a decision between a protocol-level solution (CORS) and an architectural pattern (proxy). If the target API supports CORS and is under your control, CORS is the clean, standardized solution. If the target API does not support CORS, or if you need to protect sensitive API keys from being exposed to the client, a server-side proxy is the appropriate architectural choice.

Table 4: Comparison of Cross-Origin Techniques.

Criterion	**CORS**	**JSONP**	**Server-Side Proxy**
Mechanism	Browser-server negotiation using HTTP headers. Enforced by the browser.	Client-side <script> tag injection. Server wraps response in a JavaScript callback.	Client makes a same-origin request to its own backend, which then makes a server-to-server request to the target API.
Security Profile	Secure when configured correctly. Vulnerable to misconfiguration. The server explicitly whitelists trusted origins.	**Highly Insecure.** Prone to XSS and CSRF vulnerabilities. Requires complete trust in the remote server.	Security depends on proxy implementation. Protects client-side API keys and user credentials for the target domain. Can be abused if configured as an open proxy.
Supported Methods	All HTTP methods (GET, POST, PUT, DELETE, etc.).	GET only.	All HTTP methods.
Error Handling	Robust. Provides full access to HTTP status codes and error responses.	Poor. Limited to script execution errors, no access to HTTP status codes.	Robust. The proxy can inspect the full response and relay detailed error information to the client.
Primary Use Case	The standard for all modern cross-origin communication, especially for SPAs and microservices.	Legacy applications where modern browser support is not available. **Not recommended for new development.**	Interacting with third-party APIs that do not support CORS, or when API keys must be kept secret from the client.

Troubleshooting and Conclusion

Despite its standardization, CORS remains a frequent source of frustration for web developers. The opaque nature of the errors from the perspective of JavaScript, combined with the complex interplay of headers and request types, can make debugging a challenging process. However, by adopting a systematic approach and leveraging browser developer tools, most CORS issues can be diagnosed and resolved effectively.

A Systematic Guide to Debugging Common CORS Errors

When a CORS request fails, the JavaScript code (e.g., in a catch block of a fetch promise) typically receives only a generic TypeError: Failed to fetch, without any details about the underlying cause. This is a deliberate security feature to prevent a script from gleaning information about a cross-origin resource it is not authorized to access. The key to debugging is to look beyond the script and into the browser's developer console and network tab.

Step 1: Inspect the Browser Developer Console The console is the primary source of truth for diagnosing CORS errors. Browsers provide detailed messages here that explain precisely why a request was blocked. Common error messages include:

- **Access-Control-Allow-Origin' header is missing**: This is the most common error. It means the server's response to the cross-origin request did not include the necessary Access-Control-Allow-Origin header. The solution is purely server-side: the server must be configured to send this header for the requesting origin.
- **Access-Control-Allow-Origin' header does not match '...'**: The server did send the header, but its value does not match the origin of the requesting client. This could be due to a typo in the server's whitelist, a mismatch in scheme (http vs. https), or a port number discrepancy. The server configuration must be updated to include the correct client origin.
- **Credential is not supported if the CORS header 'Access-Control-Allow-Origin' is '*'**: The client-side code requested that credentials be sent (withCredentials: true), but the server responded with the wildcard (*) origin. This is forbidden for security reasons. The server must be configured to respond with the specific client origin instead of the wildcard.
- **Response to preflight request doesn't pass access control check**: This indicates that the initial OPTIONS request (the preflight) failed. The actual request was never sent. This can happen if the server does not respond to OPTIONS requests at all, returns a server error (5xx), or responds without the correct Access-Control-Allow-Methods or

Access-Control-Allow-Headers that permit the actual request.

Step 2: Analyze the Network Tab The Network tab in the developer tools provides a detailed view of the entire request-response cycle.

- **Identify the Failing Request:** Locate the network request that is failing. It will often be highlighted in red.
- **Check for a Preflight Request:** If the request is non-simple, look for an OPTIONS request immediately preceding it. If the OPTIONS request failed (e.g., received a 404 or 500 status), that is the root cause of the problem. The server must be configured to handle OPTIONS requests on that endpoint.
- **Inspect Headers:**
  - For the request, verify that the Origin header is being sent with the expected value.
  - For the response (either to the OPTIONS or the actual request), meticulously check the Access-Control-* headers. Ensure they are present, spelled correctly, and contain the expected values.

Step 3: Use External Tools Tools like curl or Postman can be used to make requests directly to the API endpoint, bypassing the browser entirely. This helps isolate whether the issue is a fundamental server-side problem or specifically related to the CORS negotiation. If a curl request succeeds but the browser request fails, the issue is almost certainly in the server's CORS header configuration.

Concluding Thoughts: Balancing Interoperability and Security

Cross-Origin Resource Sharing is a testament to the web's evolution. It addresses the critical need for interoperability in a distributed digital ecosystem while attempting to uphold the foundational security guarantees of the Same-Origin Policy. It is a carefully designed compromise, enabling the rich, API-driven experiences of modern SPAs and microservice architectures that would otherwise be impossible.

However, the power and flexibility of CORS come with significant responsibility. The security of the protocol is not inherent; it is a direct result of diligent and correct server-side implementation. As this analysis has shown, vulnerabilities do not typically arise from flaws in the CORS standard itself but from its misuse—overly permissive policies, flawed validation logic, and a misunderstanding of its relationship with other security mechanisms like CSRF protection.

For developers and security professionals, a deep and nuanced understanding of the SOP, the mechanics of preflight requests, and the precise function of each HTTP header is not an academic exercise but an essential prerequisite for building secure and functional web applications. Ultimately, CORS provides the tools for servers to make informed trust decisions. The onus is on the implementers to use those tools wisely, crafting policies that adhere to the principle of least privilege and successfully balancing the modern web's demand for open communication with the timeless need for robust security.

WORKS CITED

1. Same-origin policy | Articles | web.dev, https://web.dev/articles/same-origin-policy
   Same-origin policy - Security | MDN, https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
2. Understanding CORS and Same Origin Policy in Web Security - DEV Community, https://dev.to/burakboduroglu/understanding-cors-and-same-origin-policy-in-web-security-54hm
3. Same-origin policy - Wikipedia, https://en.wikipedia.org/wiki/Same-origin_policy
   Understanding Same Origin Policy - Debajyati's Blogs, https://debajyatidey.hashnode.dev/demystifying-same-origin-policy-in-simple-words
4. Demystifying Cross-Origin Resource Sharing (CORS) on Web ..., https://amanexplains.com/demystifying-cross-origin-resource-sharing-on-web/
   Same-origin policy - Glossary | MDN - Mozilla,
5. https://developer.mozilla.org/en-US/docs/Glossary/Same-origin_policy
6. What is CORS? - Cross-Origin Resource Sharing Explained - AWS,
7. https://aws.amazon.com/what-is/cross-origin-resource-sharing/
8. What is CORS? A Complete Guide to Cross-Origin Resource Sharing - StackHawk, https://www.stackhawk.com/blog/what-is-cors/
9. Understanding Cross-Origin Resource Sharing (CORS) - SuperTokens,
10. https://supertokens.com/blog/what-is-cross-origin-resource-sharing
11. Cross-Origin Resource Sharing (CORS) in Web Development - Ramotion, https://www.ramotion.com/blog/what-is-cors-in-web-development/
12. What is CORS (cross-origin resource sharing)? Tutorial & Examples | Web Security Academy - PortSwigger, https://portswigger.net/web-security/cors
13. CORS OriginHeaderScrutiny - OWASP Foundation, https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny
14. Understanding CORS and Preflight Requests in APIs | by Bale - Medium,

https://medium.com/@bloodturtle/understanding-cors-and-preflight-requests-in-apis-e088ae13b 417

15. Cross-origin resource sharing - Wikipedia, https://en.wikipedia.org/wiki/Cross-origin_resource_sharing
16. Types of Cross-Origin HTTP Requests - NI, https://www.ni.com/docs/en-US/bundle/g-web-development/page/types-of-cors-requests.html
17. Cross-Origin Resource Sharing (CORS) - HTTP - MDN, https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS
18. Improve Single-Page Application (SPA) Performance with a Same Domain policy using Amazon CloudFront | Networking & Content Delivery, https://aws.amazon.com/blogs/networking-and-content-delivery/improve-single-page-application-spa-performance-with-a-same-domain-policy-using-amazon-cloudfront/
19. What is the motivation behind the introduction of preflight CORS requests? - Stack Overflow, https://stackoverflow.com/questions/15381105/what-is-the-motivation-behind-the-introduction-of- preflight-cors-requests
20. Preflight request - Glossary - MDN - Mozilla, https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request
21. CORS, Preflight Requests, and Common Cross-Origin Issues - DEV Community, https://dev.to/thesanjeevsharma/cors-preflight-requests-and-common-cross-origin-issues-129n Access-Control-Allow-Credentials header - HTTP - MDN,
22. https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Access-Control-Allow- Credentials
23. XMLHttpRequest: withCredentials property - Web APIs - MDN - Mozilla, https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/withCredentials .Access-Control-Allow-Credentials header - HTTP | MDN,
24. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Credentials
25. When should I really set "Access-Control-Allow-Credentials" to "true" in my response headers? - Stack Overflow, https://stackoverflow.com/questions/45004354/when-should-i-really-set-access-control-allow-cre dentials-to-true-in-my-resp
26. Access-Control-Allow-Origin header - HTTP - MDN, https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Access-Control-Allow-

Origin

27. Reason: Credential is not supported if the CORS header 'Access-Control-Allow-Origin' is '*',https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS/Errors/CORSNotSupporting Credentials
28. Access-Control-Allow-Origin header with wildcard (*) value - Vulnerabilities - Acunetix, https://www.acunetix.com/vulnerabilities/web/access-control-allow-origin-header-with-wildcard-v alue/
29. Testing Cross Origin Resource Sharing - WSTG - v4.1 | OWASP Foundation, https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Test ing/11-Client_Side_Testing/07-Testing_Cross_Origin_Resource_Sharing
30. HTML5 Security - OWASP Cheat Sheet Series, https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html
31. Lab: CORS vulnerability with trusted null origin | Web Security Academy - PortSwigger, https://portswigger.net/web-security/cors/lab-null-origin-whitelisted-attack
32. Cross-origin resource sharing - PortSwigger, https://portswigger.net/kb/issues/00200600_cross-origin-resource-sharing
33. Cross-origin resource sharing: all subdomains trusted - PortSwigger, https://portswigger.net/kb/issues/00200603_cross-origin-resource-sharing-all-subdomains-truste d
34. Cross-site request forgery - Wikipedia, https://en.wikipedia.org/wiki/Cross-site_request_forgery
35. What is CSRF (Cross-site request forgery)? Tutorial & Examples | Web Security Academy, https://portswigger.net/web-security/csrf
36. Cross-site request forgery (CSRF) - Security - MDN - Mozilla, https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/CSRF
37. Cross-origin resource sharing: arbitrary origin trusted - PortSwigger, https://portswigger.net/kb/issues/00200601_cross-origin-resource-sharing-arbitrary-origin-truste d
38. How Would You Manage CORS in a Production Express.js Application? - Arunangshu Das, https://arunangshudas.medium.com/how-would-you-manage-cors-in-a-production-

express-js-ap plication-45a1138dd6df

39. CORS on Apache - enable cross-origin resource sharing, https://enable-cors.org/server_apache.html
How to Enable CORS in Apache Web Server?
40. GeeksforGeeks, https://www.geeksforgeeks.org/websites-apps/how-to-enable-cors-in-apache-web-server/
41. How to allow Cross domain request in apache2 - Stack Overflow, https://stackoverflow.com/questions/29150384/how-to-allow-cross-domain-request-in-apache2
42. CORS on Nginx - enable cross-origin resource sharing, https://enable-cors.org/server_nginx.html
43. CORS in Nginx: Configuration Guide for Enhanced Security - Ercan Ermis, https://ercanermis.com/cors-in-nginx-configuration-guide-for-enhanced-security/
44. How to Configure CORS in Node.js With Express - DEV Community, https://dev.to/speaklouder/how-to-configure-cors-in-nodejs-with-express-11h
45. Node.js CORS Guide: Enable & Fix CORS in Node.js - StackHawk, https://www.stackhawk.com/blog/nodejs-cors-guide-what-it-is-and-how-to-enable-it/
46. Express cors middleware, https://expressjs.com/en/resources/middleware/cors.html
47. Security implications of cross-origin resource sharing (CORS) in Node.js - Snyk, https://snyk.io/blog/security-implications-cors-node-js/
48. CORS, Cache poisoning and the Vary HTTP header - Pixelite, https://www.pixelite.co.nz/article/cors-caching-and-the-vary-http-header/
49. CORS and Vary - text/plain, https://textslashplain.com/2018/08/02/cors-and-vary/
50. Why isn't 'Vary: Origin' response set on a CORS miss? - Stack Overflow, https://stackoverflow.com/questions/25329405/why-isnt-vary-origin-response-set-on-a-cors-miss
51. Vary: origin response header and CORS exploitation - Information Security Stack Exchange, https://security.stackexchange.com/questions/151590/vary-origin-response-header-and-cors-ex ploitation
SPA (Single-page application) - Glossary - MDN,
52. https://developer.mozilla.org/en-US/docs/Glossary/SPA
53. Use OAuth with CORS to connect a SPA - Power Apps | Microsoft Learn, https://learn.microsoft.com/en-us/power-apps/developer/data-platform/oauth-cross-origin-resour ce-sharing-connect-single-page-application

54. Single-page applications and CORS - Docusign Developer, https://developers.docusign.com/platform/single-page-applications-cors/
55. Using OAuth for Single Page Applications | Best Practices - Curity, https://curity.io/resources/learn/spa-best-practices/
56. CORS vs. JSONP: When to Use Each Technique - CorsProxy.io, https://corsproxy.io/blog/cors-vs-jsonp/
57. What is JSONP, and why was it created? - Stack Overflow, https://stackoverflow.com/questions/2067472/what-is-jsonp-and-why-was-it-created
58. CORS & JSONP | Socrata - Data & Insights, https://dev.socrata.com/docs/cors-and-jsonp
59. So, JSONP or CORS? - Stack Overflow, https://stackoverflow.com/questions/12296910/so-jsonp-or-cors
60. Understanding JSON, JSONP, CORS and bypassing CORS with JSONP | by Kunal Tandon | Developer's Arena | Medium, https://medium.com/developers-arena/understanding-json-jsonp-cors-and-bypassing-cors-with-j sonp-fa5f0cc4edd4
61. JSONP - Wikipedia, https://en.wikipedia.org/wiki/JSONP
62. web application - Security risks with JSONP?, https://security.stackexchange.com/questions/23438/security-risks-with-jsonp
63. What are CORS proxies, and when are they safe? - HTTP Toolkit, https://httptoolkit.com/blog/cors-proxies/
64. Fixing CORS Errors — How to Build a Proxy Server to Handle Cross-Origin Requests,
65. https://jakemccambley.medium.com/fixing-cors-errors-when-working-with-3rd-party-apis-a69dc5 474804
    http - How do CORS proxy websites work? - Information Security ..., https://security.stackexchange.com/questions/191737/how-do-cors-proxy-websites-work
66. CORS Web request through a proxy is security breach? - Stack Overflow, https://stackoverflow.com/questions/48788171/cors-web-request-through-a-proxy-is-security-breach
67. CORS errors - HTTP | MDN - Mozilla, https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS/Errors
68. Way to debug CORS errors - Stack Overflow, https://stackoverflow.com/questions/20032037/way-to-debug-cors-errors

69. Understanding CORS errors: Key causes and effective solutions - Contentstack, https://www.contentstack.com/blog/tech-talk/understanding-cors-errors-key-causes-and-effectiv e-solutions
70. Four Common CORS Errors and How to Fix Them - Descope, https://www.descope.com/blog/post/cors-errors
71. I got a CORS error, now what? - DEV Community, https://dev.to/authress/i-got-a-cors-error-now-what-hpb
72. How to fix CORS errors: A comprehensive guide for web developers - Contentstack, https://www.contentstack.com/blog/strategy/how-to-fix-cors-errors-a-comprehensive-guide-for-w eb-developers
73. Understanding CORS and CSRF: A Guide for Spring Security | by Suresh - Medium,
74. https://medium.com/@CodeWithTech/understanding-cors-and-csrf-a-guide-for-spring-security-fe b34b81a3a4