基於安全考量，從 1995 年 Netscape 開始，瀏覽器都實現同源策略（Same-origin policy），但是，隨著前端應用的多樣化，跨域請求的需求無可迴避，於是，後來 CORS（Cross-Origin Resource Sharing）成了規範。

在 XMLHttpRequest Level 1 規範之後，XMLHttpRequest 本身就可以進行跨域請求了，Fetch API 預設也支援跨域請求，它們實現了 W3C 正式的 CORS 規範，其中，不單只是規範瀏覽器處理跨域請求的方式，也規範了伺服端可控制的項目，像是允許的來源、請求方法、可否發送 Cookie、可取得的回應標頭，甚至回應有效期限等。

支援 CORS 的瀏覽器，會在跨域請求時，自動處理細節，然而依舊必須知道，CORS 將跨域請求分為：簡單（Simple），以及帶預檢（with Preflight）等兩種。在簡單請求中，方法只能是 HEAD、GET、POST，而且，可用標頭，只能是 Accept、Accept-Language、Content-Language、Last-Event-ID，以及 Content-Type；而 Content-Type 也只允許三個值 application/x-www-form-urlencoded、multipart/form-data、text/plain，此外都算入帶預檢的請求。

而且，簡單請求之所以「簡單」，是因為瀏覽器會直接發出跨域請求，並帶上 Origin 標頭表明來源，供伺服端用以判斷是否允許請求，若允許的話，可透過 Access-Control-開頭的標頭來控制回應，最常見的就是 Access-Control-Allow-Origin（後文簡稱 ACAO），若設定為與 Origin 相同或者是「*」（不限制來源），瀏覽器就能允許指令稿拿到回應，否則就拋出錯誤，伺服端還可以使用 Access-Control-Allow-Credentials，來表示瀏覽器是否可發送 Cookie、Access-Control-Expose-Headers，決定瀏覽器可拿到的回應標頭。

若是另一種「帶預檢」的請求，瀏覽器會先使用 OPTIONS 來試探伺服器，並附上 Origin 及 Access-Control-Request-Method、Access-Control-Request-Headers 標頭，表明使用的請求方法，以及自訂的標頭，伺服端若允許請求，除了 ACAO 之外，還會附上 Access-Control-Allow-Methods、Access-Control-Allow-Headers 標頭，表示允許的方法及自訂標頭，之後，瀏覽器可依此決定伺服端是否同意跨域請求——若否，就拋出錯誤；若是，接下來就是正式進行跨域請求，方式就與簡單請求相同了。

基本上，同源策略是為了瀏覽器安全性，JSONP 則是直接繞過了瀏覽器，除非是為了相容於老舊的瀏覽器，否則不建議使用。然而，CORS 雖是正式規範，瀏覽器也會自動處理細節，但並非保證安全無虞。

由於 CORS 規範中，ACAO 只允許設定為「*」或者是單一網域來源，若伺服端想要支援多個來源，就必須檢查 Origin 內容，決定是否允許並動態產生 ACAO 的值，Origin 檢查規則若設計不良，例如，單純檢查前置或後置，就很容易被繞過，或者未過濾惡意符號，有機會造成某些組合可繞過 Origin 的檢查規則。 單就繞過 Origin 檢查規則來說，就可能達成攻擊內網的條件。舉例來說，若 Access-Control-Allow-Credentials 設定為 true，攻擊者就可能藉由發送 Cookie（例如將 XMLHttpRequest 的 withCredentials 設為 true），進一步跨域取得使用者機密資料，或者執行 CSRF 攻擊；ACAO 設定為「*」時，不會允許發送 Cookie，然而，在動態生成其他網域值的情況下，非必要的話，就別將 Access-Control-Allow-Credentials 設為 true。

另外，使用 CORS 須留意一下快取的問題。因為，瀏覽器可能對同樣 URL 的資源進行快取，而快取內容會包含回應標頭，在跨域時，就可能發生 ACAO 設定不同，然而，實際使用的是相同 URL 資源，結果卻直接從快取中取得，因而造成請求失敗等問題。