Extend the Browser

- What are Extensions?
- Get Started Tutorial
- Overview
- Developer Guide
  - Reach Peak Performance
  - Protect User Privacy
  - Stay Secure
  - Debugging
  - OAuth
  - Accessibility
  - Content Security Policy
  - **Cross-Origin XHR**
  - Internationalization
  - Message Passing
  - Native Messaging
  - Match Patterns
  - Extension Quality Guidelines FAQ
- Samples
- Help

Web Store Publishing and Distribution

Mobile Chrome

Cross-Origin XMLHttpRequest

Regular web pages can use the **XMLHttpRequest** object to send and receive data from remote servers, but they're limited by the **same origin policy**. Extensions aren't so limited. An extension can talk to remote servers outside of its origin, as long as it first requests cross-origin permissions.

Extension origin

Each running extension exists within its own separate security origin. Without requesting additional privileges, the extension can use XMLHttpRequest to get resources within its installation. For example, if an extension contains a JSON configuration file called `config.json`, in a `config_resources` folder, the extension can retrieve the file's contents like this:

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = handleStateChange; // Implemented elsewhere.
xhr.open("GET", chrome.extension.getURL('/config_resources/config.json'), true);
xhr.send();
```

If the extension attempts to use a security origin other than itself, say https://www.google.com, the browser disallows it unless the extension has requested the appropriate cross-origin permissions.

Requesting cross-origin permissions

By adding hosts or host match patterns (or both) to the **permissions** section of the **manifest** file, the extension can request access to remote servers outside of its origin.

```
{
  "name": "My extension",
  ...
  "permissions": [
    "https://www.google.com/"
  ],
  ...
}
```

Cross-origin permission values can be fully qualified host names, like these:

- "https://www.google.com/"
- "https://www.gmail.com/"

Or they can be match patterns, like these:

- "https://*.google.com/"
- "https://*/"

A match pattern of "https://*/" allows HTTPS access to all reachable domains. Note that here, match patterns are similar to **content script match patterns**, but any path information following the host is ignored.

Also note that access is granted both by host and by scheme. If an extension wants both secure and non-secure HTTP access to a given host or set of hosts, it must declare the permissions separately:

```
"permissions": [
  "http://www.google.com/",
  "https://www.google.com/"
]
```

Security considerations

When using resources retrieved via XMLHttpRequest, your background page should be careful not to fall victim to **cross-site scripting**. Specifically, avoid using dangerous APIs such as the below:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.example.com/data.json", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    // WARNING! Might be evaluating an evil script!
    var resp = eval("(" + xhr.responseText + ")");
    ...
  }
}
xhr.send();
```

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.example.com/data.json", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    // WARNING! Might be injecting a malicious script!
    document.getElementById("resp").innerHTML = xhr.responseText;
    ...
  }
}
xhr.send();
```

Instead, prefer safer APIs that do not run scripts:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.example.com/data.json", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
```

```
    // JSON.parse does not evaluate the attacker's scripts.
    var resp = JSON.parse(xhr.responseText);
  }
}
xhr.send();
```

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.example.com/data.json", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    // innerText does not let the attacker inject HTML elements.
    document.getElementById("resp").innerText = xhr.responseText;
  }
}
xhr.send();
```

Additionally, be especially careful of resources retrieved via HTTP. If your extension is used on a hostile network, an network attacker (aka a **"man-in-the-middle"**) could modify the response and, potentially, attack your extension. Instead, prefer HTTPS whenever possible.

Interaction with Content Security Policy

If you modify the default **Content Security Policy** for apps or extensions by adding a `content_security_policy` attribute to your manifest, you'll need to ensure that any hosts to which you'd like to connect are allowed. While the default policy doesn't restrict connections to hosts, be careful when explicitly adding either the `connect-src` or `default-src` directives.