

# Cross-origin Service Workers: Experimenting with Foreign Fetch



By Jeff Posnick

Web DevRel @ Google

**Warning:** The information in this post is out of date. Foreign fetch is no longer available for testing in Chrome, and has been removed from the service worker specification.

## Background

Service workers give web developers the ability to respond to network requests made by their web applications, allowing them to continue working even while offline, fight lie-fi, and implement complex cache interactions like stale-while-revalidate. But service workers have historically been tied to a specific origin—as the owner of a web app, it's your responsibility to write and deploy a service worker to intercept all the network requests your web app makes. In that model, each service worker is responsible for handling even cross-origin requests, for example to a third-party API or for web fonts.

What if a third-party provider of an API, or web fonts, or other commonly used service had the power to deploy their own service worker that got a chance to handle requests made by *other* origins to their origin? Providers could implement their own custom networking logic, and take advantage of a single, authoritative cache instance for storing their responses. Now, thanks to foreign fetch, that type of third-party service worker deployment is a reality.

Deploying a service worker that implements foreign fetch makes sense for any provider of a service that's accessed via HTTPS requests from browsers—just think about scenarios in which you could provide a network-independent version of your service, in which browsers could take advantage of a common resource cache. Services that could benefit from this include, but are not limited to:

- API providers with RESTful interfaces
- Web font providers
- Analytics providers
- Image hosting providers

- Generic content delivery networks

Imagine, for instance, that you're an analytics provider. By deploying a foreign fetch service worker, you can ensure that all requests to your service that fail while a user is offline are queued and replayed once connectivity returns. While it's been possible for a service's clients to implement similar behavior via first-party service workers, requiring each and every client to write bespoke logic for your service is not as scalable as relying on a shared foreign fetch service worker that you deploy.

## Prerequisites

### Origin Trial token

Foreign fetch is still considered experimental. In order to keep from prematurely baking this design in before it's fully specified and agreed upon by browser vendors, it's been implemented in Chrome 54 as an Origin Trial. As long as foreign fetch remains experimental, to use this new feature with the service you host, you'll need to request a token that's scoped to your service's specific origin. The token should be included as an HTTP response header in all cross-origin requests for resources that you want to handle via foreign fetch, as well as in the response for your service worker JavaScript resource:

Origin-Trial: token\_obtained\_from\_signup



The trial will end in March 2017. By that point, we expect to have figured out any changes necessary to stabilize the feature, and (hopefully) enable it by default. If foreign fetch is not enabled by default by that time, the functionality tied to existing Origin Trial tokens will stop working.

To facilitate experimenting with foreign fetch prior to registering for an official Origin Trial token, you can bypass the requirement in Chrome for your local computer by going to `chrome://flags/#enable-experimental-web-platform-features` and enabling the "Experimental Web Platform features" flag. Please note that this needs to be done in every instance of Chrome that you want to use in your local experimentations, whereas with an Origin Trial token the feature will be available to all of your Chrome users.

## HTTPS

As with all service worker deployments, the web server you use for serving both your resources and your service worker script needs to be accessed via HTTPS. Additionally, foreign fetch interception only applies to requests that originate from pages hosted on

secure origins, so the clients of your service need to use HTTPS to take advantage of your foreign fetch implementation.

## Using Foreign Fetch

With the prerequisites out of the way, let's dive into the technical details needed to get a foreign fetch service worker up and running.

### Registering your service worker

The first challenge that you're likely to bump into is how to register your service worker. If you've worked with service workers before, you're probably familiar with the following:

```
// You can't do this!  
if ('serviceWorker' in navigator) {  
  navigator.serviceWorker.register('service-worker.js');  
}
```



This JavaScript code for a first-party service worker registration makes sense in the context of a web app, triggered by a user navigating to a URL you control. But it's not a viable approach to registering a third-party service worker, when the only interaction browser will have with your server is requesting a specific subresource, not a full navigation. If the browser requests, say, an image from a CDN server that you maintain, you can't prepend that snippet of JavaScript to your response and expect that it will be run. A different method of service worker registration, outside the normal JavaScript execution context, is required.

The solution comes in the form of an HTTP header that your server can include in any response:

```
Link: </service-worker.js>; rel="serviceworker"; scope="/"
```



Let's break down that example header into its components, each of which is separated by a ; character.

- `</service-worker.js>` is required, and is used to specify the path to your service worker file (replace `/service-worker.js` with the appropriate path to your script). This corresponds directly to the scriptURL string that would otherwise be passed as the first parameter to `navigator.serviceWorker.register()`. The value needs to be enclosed in `<>` characters (as required by the Link header specification), and if a

relative rather than absolute URL is provided, it will be interpreted as being relative to the location of the response.

- `rel="serviceworker"` is also required, and should be included without any need for customization.
- `scope=` is an optional scope declaration, equivalent to the `options.scope` string you can pass in as the second parameter to `navigator.serviceWorker.register()`. For many use cases, you're fine with using the default scope, so feel free to leave this out unless you know that you need it. The same restrictions around maximum allowed scope, along with the ability to relax those restrictions via the `Service-Worker-Allowed` header, apply to `Link` header registrations.

Just like with a "traditional" service worker registration, using the `Link` header will install a service worker that will be used for the *next* request made against the registered scope. The body of the response that includes the special header will be used as-is, and is available to the page immediately, without waiting for the foreign service worker to finish installation.

Remember that foreign fetch is currently implemented as an Origin Trial, so alongside your `Link` response header, you'll need to include a valid `Origin-Trial` header as well. The minimum set of response headers to add in order to register your foreign fetch service worker is

```
Link: </service-worker.js>; rel="serviceworker"  
Origin-Trial: token_obtained_from_signup
```



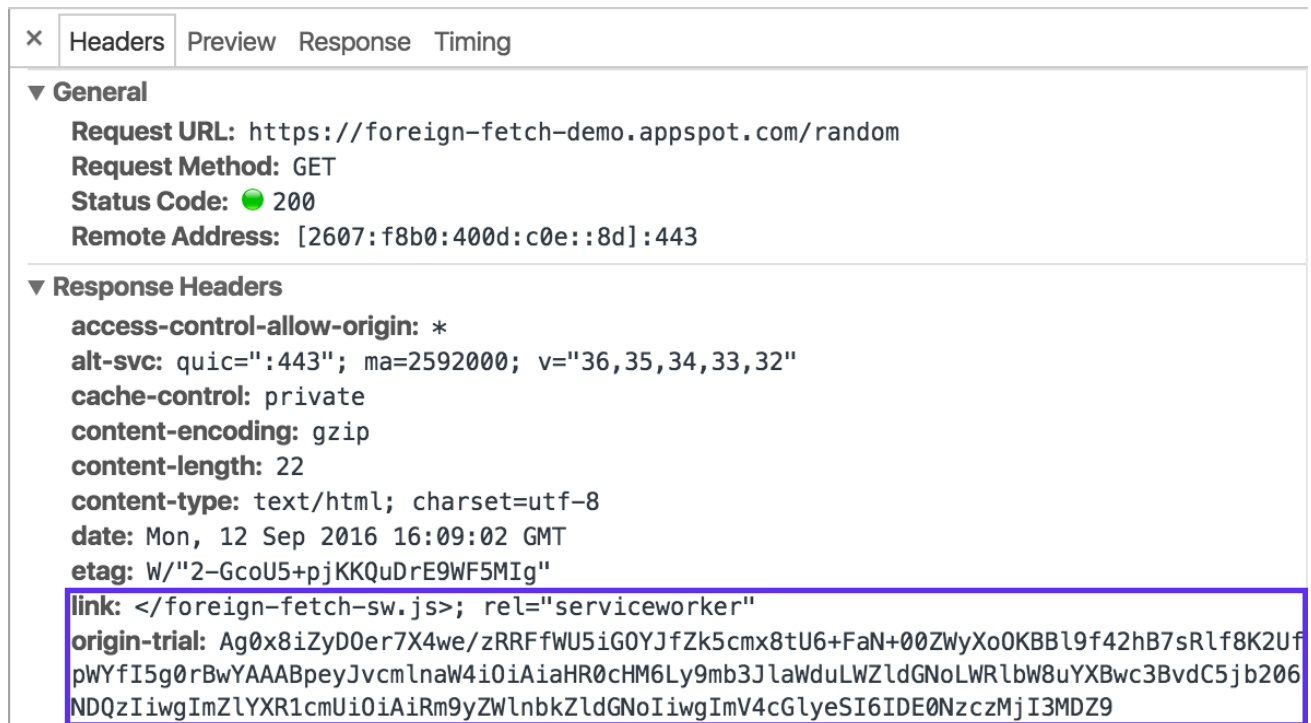
**Note:** Astute readers of the service worker specification may have noticed another means of performing service worker registration, via a `<link rel="serviceworker">` DOM element. Support for `<link>`-based registration in Chrome is currently controlled by the same Origin Trial as the `Link` header, so it is not yet enabled by default. `<link>`-based registration has the same limitations as JavaScript-based registration when it comes to foreign fetch registration, so for the purposes of this article, the `Link` header is what you should be using.

## Debugging registration

During development, you'll probably want to confirm that your foreign fetch service worker is properly installed and processing requests. There are a few things you can check in Chrome's Developer Tools to confirm that things are working as expected.

**Are the proper response headers being sent?**

In order to register the foreign fetch service worker, you need to set a Link header on a response to a resource hosted on your domain, as described earlier in this post. During the Origin Trial period, and assuming you don't have `chrome://flags/#enable-experimental-web-platform-features` set, you also need to set a `Origin-Trial` response header. You can confirm that your web server is setting those headers by looking at the entry in the Network panel of DevTools:



× Headers Preview Response Timing

▼ General

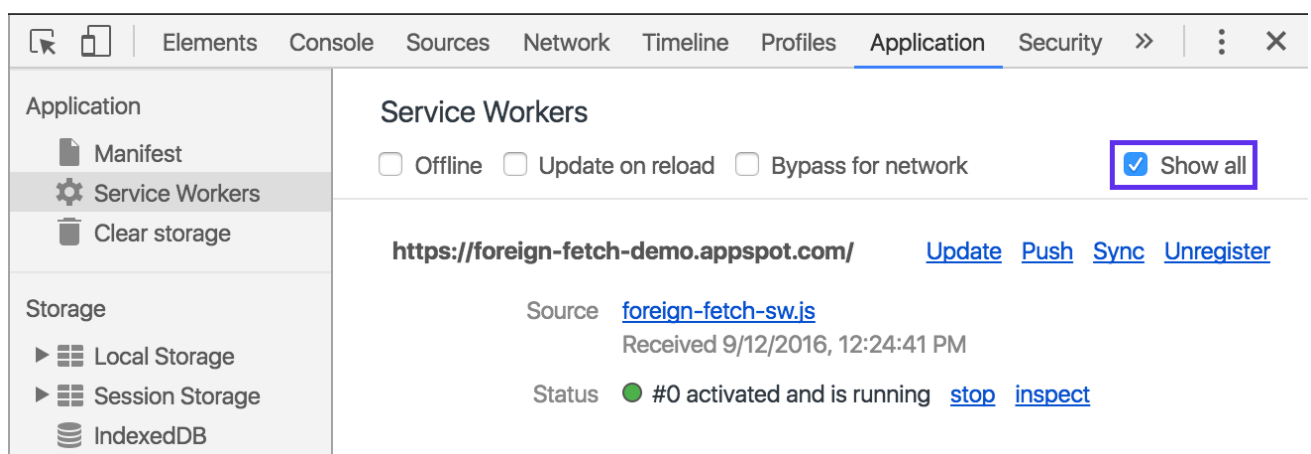
**Request URL:** https://foreign-fetch-demo.appspot.com/random  
**Request Method:** GET  
**Status Code:** 200  
**Remote Address:** [2607:f8b0:400d:c0e::8d]:443

▼ Response Headers

**access-control-allow-origin:** \*  
**alt-svc:** quic=":443"; ma=2592000; v="36,35,34,33,32"  
**cache-control:** private  
**content-encoding:** gzip  
**content-length:** 22  
**content-type:** text/html; charset=utf-8  
**date:** Mon, 12 Sep 2016 16:09:02 GMT  
**etag:** W/"2-GcoU5+pjKKQuDrE9WF5MIg"  
**link:** </foreign-fetch-sw.js>; rel="serviceworker"  
**origin-trial:** Ag0x8iZyD0er7X4we/zRRfFWU5iG0YJfZk5cmx8tU6+FaN+00ZWYXo0KBB19f42hB7sRlf8K2UfpWYfI5g0rBwYAAABpeyJvcmlnaW4iOiAiaHR0cHM6Ly9mb3JlaWduLWZldGNoLWRlbW8uYXBwc3BvdC5jb206NDQzIiwgImZlYXR1cmUiOiAiRm9yZWlnbkZldGNoIiwgImV4cGlyeSI6IDE0NzczMjI3MDZ9

## Is the Foreign Fetch service worker properly registered?

You can also confirm the underlying service worker registration, including its scope, by looking at the full list of service workers in the Application panel of DevTools. Make sure to select the "Show all" option, since by default, you'll only see service workers for the current origin.



Application

Manifest  
Service Workers  
Clear storage

Storage

Local Storage  
Session Storage  
IndexedDB

Service Workers

☐ Offline ☐ Update on reload ☐ Bypass for network ☒ Show all

https://foreign-fetch-demo.appspot.com/ [Update](#) [Push](#) [Sync](#) [Unregister](#)

Source [foreign-fetch-sw.js](#)  
Received 9/12/2016, 12:24:41 PM

Status ● #0 activated and is running [stop](#) [inspect](#)

## The install event handler

Now that you've registered your third-party service worker, it will get a chance to respond to the `install` and `activate` events, just like any other service worker would. It can take advantage of those events to, for example, populate caches with required resources during the `install` event, or prune out-of-date caches in the `activate` event.

Beyond normal `install` event caching activities, there's an additional step that's **required** inside your third-party service worker's `install` event handler. Your code needs to call `registerForeignFetch()`, as in the following example:

```
self.addEventListener('install', event => {  
  event.registerForeignFetch({  
    scopes: [self.registration.scope], // or some sub-scope  
    origins: ['*'] // or ['https://example.com']  
  });  
});
```



There are two configuration options, both required:

- `scopes` takes an array of one or more strings, each of which represents a scope for requests that will trigger a `foreignfetch` event. *But wait, you may be thinking, I've already defined a scope during service worker registration!* That's true, and that overall scope is still relevant—each scope that you specify here must be either equal to or a sub-scope of the service worker's overall scope. The additional scoping restrictions here allow you to deploy an all-purpose service worker that can handle both first-party `fetch` events (for requests made from your own site) and third-party `foreignfetch` events (for requests made from other domains), and make it clear that only a subset of your larger scope should trigger `foreignfetch`. In practice, if you're deploying a service worker dedicated to handling only third-party, `foreignfetch` events, you're just going to want to use a single, explicit scope that's equal to your service worker's overall scope. That's what the example above will do, using the value `self.registration.scope`.
- `origins` also takes an array of one or more strings, and allows you to restrict your `foreignfetch` handler to only respond to requests from specific domains. For example, if you explicitly whitelist `'https://example.com'`, then a request made from a page hosted at `https://example.com/path/to/page.html` for a resource served from your foreign fetch scope will trigger your foreign fetch handler, but requests made from `https://random-domain.com/path/to/page.html` won't trigger your handler. Unless you have a specific reason to only trigger your foreign fetch logic for a subset of remote origins, you can just specify `'*'` as the only value in the array, and all origins will be whitelisted.

## The foreignfetch event handler

Now that you've installed your third-party service worker and it's been configured via `registerForeignFetch()`, it will get a chance to intercept cross-origin subresource requests to your server that fall within the foreign fetch scope.

**Note:** There's an additional restriction in Chrome's current implementation: only GET, POST, or HEAD requests that contain only CORS-safelisted headers are eligible for foreign fetch. This restriction is not part of the foreign fetch specification and may be relaxed in future versions of Chrome.

In a traditional, first-party service worker, each request would trigger a fetch event that your service worker had a chance to respond to. Our third-party service worker is given a chance to handle a slightly different event, named `foreignfetch`. Conceptually, the two events are quite similar, and they give you the opportunity to inspect the incoming request, and optionally provide a response to it via `respondWith()`:

```
self.addEventListener('foreignfetch', event => {  
  // Assume that requestLogic() is a custom function that takes  
  // a Request and returns a Promise which resolves with a Response.  
  event.respondWith(  
    requestLogic(event.request).then(response => {  
      return {  
        response: response,  
        // Omit to origin to return an opaque response.  
        // With this set, the client will receive a CORS response.  
        origin: event.origin,  
        // Omit headers unless you need additional header filtering.  
        // With this set, only Content-Type will be exposed.  
        headers: ['Content-Type']  
      };  
    })  
  );  
});
```



Despite the conceptual similarities, there are a few differences in practice when calling `respondWith()` on a `ForeignFetchEvent`. Instead of just providing a Response (or Promise that resolves with a `Response`) to `respondWith()`, like you do with a FetchEvent, you need to pass a `Promise` that resolves with an Object with specific properties to the `ForeignFetchEvent`'s `respondWith()`:

- `response` is required, and must be set to the `Response` object that will be returned to the client that made the request. If you provide anything other than a valid `Response`, the client's request will be terminated with a network error. Unlike when calling

respondWith() inside a `fetch` event handler, you **must** provide a `Response` here, not a `Promise` which resolves with a `Response`! You can construct your response via a promise chain, and pass that chain as the parameter to `foreignfetch`'s `respondWith()`, but the chain must resolve with an `Object` that contains the `response` property set to a `Response` object. You can see a demonstration of this in the code sample above.

- `origin` is optional, and it's used to determine whether or not the response that's returned is opaque. If you leave this out, the response will be opaque, and the client will have limited access to the response's body and headers. If the request was made with mode: 'cors', then returning an opaque response will be treated as an error. However, if you specify a string value equal to the origin of the remote client (which can be obtained via `event.origin`), you're explicitly opting in to provides a CORS-enabled response to the client.
- `headers` is also optional, and is only useful if you're also specifying `origin` and returning a CORS response. By default, only headers in the CORS-safelisted response header list will be included in your response. If you need to further filter what's returned, `headers` takes a list of one or more header names, and it will use that as a whitelist of which headers to expose in the response. This allows you to opt-in to CORS while still preventing potentially sensitive response headers from being exposed directly to the remote client.

It's important to note that when the `foreignfetch` handler is run, **it has access to all the credentials and ambient authority of the origin hosting the service worker**. As a developer deploying a foreign fetch-enabled service worker, it's your responsibility to ensure that you do not leak any privileged response data that would not otherwise be available by virtue of those credentials. Requiring an opt-in for CORS responses is one step to limit inadvertent exposure, but as a developer you can explicitly make `fetch()` requests inside your `foreignfetch` handler that do not use the implied credentials via:

```
self.addEventListener('foreignfetch', event => {  
  // The new Request will have credentials omitted by default.  
  const noCredentialsRequest = new Request(event.request.url);  
  event.respondWith(  
    // Replace with your own request logic as appropriate.  
    fetch(noCredentialsRequest)  
      .catch(() => caches.match(noCredentialsRequest))  
      .then(response => ({response})),  
  );  
});
```



## Client considerations



There are some additional considerations that affect how your foreign fetch service worker handles requests made from clients of your service.

## Clients that have their own first-party service worker

Some clients of your service may already have their own first-party service worker, handling requests originating from their web app. What does this mean for your third-party, foreign fetch service worker?

The `fetch` handler(s) in a first-party service worker get the first opportunity to respond to all requests made by the web app, even if there's a third-party service worker with `foreignfetch` enabled with a scope that covers the request. But clients with first-party service workers can still take advantage of your foreign fetch service worker!

Inside a first-party service worker, using `fetch()` to retrieve cross-origin resources will trigger the appropriate foreign fetch service worker. That means code like the following can take advantage of your `foreignfetch` handler:

```
// Inside a client's first-party service-worker.js:
self.addEventListener('fetch', event => {
  // If event.request is under your foreign fetch service worker's
  // scope, this will trigger your foreignfetch handler.
  event.respondWith(fetch(event.request));
});
```



Similarly, if there are first-party fetch handlers, but they don't call `event.respondWith()` when handling requests for your cross-origin resource, the request will automatically "fall through" to your `foreignfetch` handler:

```
// Inside a client's first-party service-worker.js:
self.addEventListener('fetch', event => {
  if (event.request.mode === 'same-origin') {
    event.respondWith(localRequestLogic(event.request));
  }

  // Since event.respondWith() isn't called for cross-origin requests,
  // any foreignfetch handlers scoped to the request will get a chance
  // to provide a response.
});
```



If a first-party `fetch` handler calls `event.respondWith()` but does *not* use `fetch()` to request a resource under your foreign fetch scope, then your foreign fetch service worker will not get a chance to handle the request.

## Clients that don't have their own service worker

All clients that make requests to a third-party service can benefit when the service deploys a foreign fetch service worker, even if they aren't already using their own service worker. There is nothing specific that clients need to do in order to opt-in to using a foreign fetch service worker, as long as they're using a browser that supports it. This means that by deploying a foreign fetch service worker, your custom request logic and shared cache will benefit many of your service's clients immediately, without them taking further steps.

## Putting it all together: where clients look for a response

Taking into account the information above, we can assemble a hierarchy of sources a client will use to find a response for a cross-origin request.

1. A first-party service worker's `fetch` handler (if present)
2. A third-party service worker's `foreignfetch` handler (if present, and only for cross-origin requests)
3. The browser's [HTTP cache](#) (if a fresh response exists)
4. The network

The browser starts from the top and, depending on the service worker implementation, will continue down the list until it finds a source for the response.

## Learn more

- [Foreign fetch explainer](#)
- [Sample code and live demo](#)
- [Service worker issue tracker](#)

## Stay up to date

Chrome's implementation of the foreign fetch Origin Trial is subject to change as we address feedback from developers. We'll keep this post up to date via inline changes, and will make note the specific changes below as they happen. We'll also share information about major changes via the [@chromiumdev](#) Twitter account.

---

*Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.*

*Last updated July 2, 2018.*