Introduction to Feature Policy

By Eric Bidelman
Engineer @ Google working on web tooling: Headless Chrome, Puppeteer, Lighthouse

TL;DR

Feature Policy allows web developers to selectively enable, disable, and modify the behavior of certain APIs and web features in the browser. **It's like CSP but instead of controlling security, it controls features!**

The feature policies themselves are little opt-in agreements between developer and browser that can help foster our goals of building (and maintaining) high quality web apps.

Introduction

Building for the web is a rocky adventure. It's hard enough to build a top-notch web app that nails performance and uses all the latest best practices. It's even harder to keep that experience great over time. As your project evolves, developers come on board, new features land, and the codebase grows. That Great Experience ™ you once achieved may begin to deteriorate and UX starts to suffer! Feature Policy is designed to keep you on track.

With Feature Policy, you **opt-in to a set of "policies"** for the browser to enforce on specific features used throughout your site. These policies restrict what APIs the site can access or modify the browser's default behavior for certain features.

Here are examples of things you can do with Feature Policy:

- Change the default behavior of `autoplay` on mobile and third party videos.
- Restrict a site from using sensitive APIs like `camera` or `microphone`.
- Allow iframes to use the `fullscreen` API.
- Block the use of outdated APIs like synchronous XHR and `document.write()`.
- Ensure images are sized properly (e.g. prevent layout thrashing) and are not too big for the viewport (e.g. waste user's bandwidth).

Policies are a contract between developer and browser. They inform the browser about what the developer's intent is and thus, help keep us honest when our app tries to go off the rails and do something bad. If the site or embedded third-party content attempts to violate any of the developer's preselected rules, the browser overrides the behavior with better UX or blocks the API altogether.

Using Feature Policy

Feature Policy provides two ways to control features:

1. Through the `Feature-Policy` HTTP header.
2. With the `allow` attribute on iframes.

The biggest difference between the HTTP header and the `allow` attribute is that the `allow` attribute only controls features within an iframe. The header can control features in the main response + any iframe'd content within the page. This is because iframes inherit the policies of their parent page.

The `Feature-Policy` HTTP header

The easiest way to use Feature Policy is by sending the `Feature-Policy` HTTP header with the response of a page. The value of this header is a policy or set of policies that you want the browser to respect for a given origin:

```
Feature-Policy: <feature> <allow list origin(s)>
```

The origin allow list can take several different values:

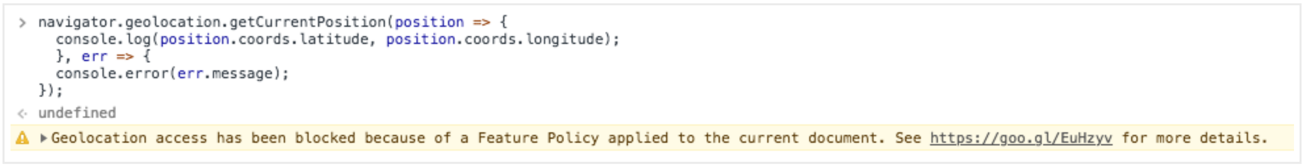
- `*`: The feature is allowed in top-level browsing contexts and in nested browsing contexts (iframes).
- `'self'`: The feature is allowed in top-level browsing contexts and same-origin nested browsing contexts. It is disallowed in cross-origin documents in nested browsing contexts.
- `'none'`: The feature is disallowed in top-level browsing contexts and disallowed in nested browsing contexts.
- `<origin(s)>`: specific origins to enable the policy for (e.g. `https://example.com`).

Example

Let's say you wanted to block all content from using the Geolocation API across your site. You can do that by sending a strict allowlist of `'none'` for the `geolocation` feature:

```
Feature-Policy: geolocation 'none'
```

If a piece of code or iframe tries to use the Geolocation API, the browser blocks it. **This is true even if the user has previously given permission to share their location**.



```
> navigator.geolocation.getCurrentPosition(position => {
    console.log(position.coords.latitude, position.coords.longitude);
    }, err => {
    console.error(err.message);
  });
<· undefined
⚠ ▶Geolocation access has been blocked because of a Feature Policy applied to the current document. See https://goo.gl/EuHzyv for more details.
```

Violating the set geolocation policy.

In other cases, it might make sense to relax this policy a bit. We can allow our own origin to use the Geolocation API but prevent third-party content from accessing it by setting `'self'` in the allow list:

```
Feature-Policy: geolocation 'self'
```

The iframe `allow` attribute

The second way to use Feature Policy is for controlling content within an `iframe`. Use the `allow` attribute to specify a policy list for embedded content:

```
<!-- Allow all browsing contexts within this iframe to use fullscreen. -->
<iframe src="https://example.com..." allow="fullscreen"></iframe>

<!-- Equivalent to: -->
<iframe src="https://example.com..." allow="fullscreen *"></iframe>

<!-- Allow only iframe content on a particular origin to access the user's locati
<iframe src="https://google-developers.appspot.com/demos/..."
        allow="geolocation https://google-developers.appspot.com"></iframe>
```

Note: Frames inherit the policy settings of their parent page. If the page and iframe both specify a policy list, the more restrictive allow list is used. See Inheritance rules.

What about the existing iframe attributes?

Some of the features controlled by Feature Policy have an existing attribute to control their behavior. For example, `<iframe allowfullscreen>` is an attribute that allows iframe content to use the `HTMLElement.requestFullscreen()` API. There's also the `allowpaymentrequest` and `allowusermedia` attributes for allowing the Payment Request API and `getUserMedia()`, respectively.

Try to **use the `allow` attribute instead of these old attributes** where possible. For cases where you need to support backwards compatibility using the `allow` attribute with an equivalent legacy attribute is perfectly fine (e.g. `<iframe allowfullscreen allow="fullscreen">`). Just note that the more restrictive policy wins. For example, the following iframe would not be allowed to enter fullscreen because `allow="fullscreen 'none'"` is more restrictive than `allowfullscreen`:

```
<!-- Blocks fullscreen access if the browser supports feature policy. -->
<iframe allowfullscreen allow="fullscreen 'none'" src="...">
```

Controlling multiple policies at once

Several features can be controlled at the same time by sending the HTTP header with a `;` separated list of policy directives:

```
Feature-Policy: unsized-media 'none'; geolocation 'self' https://example.co
```

or by sending a separate header for each policy:

```
Feature-Policy: unsized-media 'none'
Feature-Policy: geolocation 'self' https://example.com
Feature-Policy: camera *;
```

This example would do the following:

- Disallows the use of `unsized-media` for all browsing contexts.
- Disallows the use of `geolocation` for all browsing contexts except for the page's own origin and `https://example.com`.
- Allows `camera` access for all browsing contexts.

Example - setting multiple policies on an iframe

```
<!-- Blocks the iframe from using the camera and microphone
     (if the browser supports feature policy). -->
<iframe allow="camera 'none'; microphone 'none'">
```

JavaScript API

Heads up: While Chrome 60 added support for the `Feature-Policy` HTTP header and the `allow` attribute on iframes, the JavaScript API is still being fleshed out and is likely to change as it goes through the standardization process. You can enable the API using the `--enable-experimental-web-platform-features` flag in `chrome:flags`.

Feature Policy includes a small JavaScript API to allow client-side code to determine what features are allowed by a page or frame. You can access its goodies under `document.policy` for the main document or `frame.policy` for iframes.

Example

The example below illustrates the results of sending a policy of `Feature-Policy: geolocation 'self'` on the site `https://example.com`:

```
/* @return {Array<string>} List of feature policies allowed by the page. */
document.policy.allowedFeatures();
// → ["geolocation", "midi",  "camera", "usb", "autoplay",...]

/* @return {boolean} True if the page allows the 'geolocation' feature. */
document.policy.allowsFeature('geolocation');
// → true

/* @return {boolean} True if the provided origin allows the 'geolocation' feature
document.policy.allowsFeature('geolocation', 'https://google-developers.appspot.c
// → false

/* @return {Array<string>} List of origins (used throughout the page) that are
   allowed to use the 'geolocation' feature. */
document.policy.getAllowlistForFeature('geolocation');
// → ["https://example.com"]
```

List of policies

So what features can be controlled through Feature Policy?

Right now, there's a lack of documentation on what policies are implemented and how to use them. The list will also grow over time as different browsers adopt the spec and implement various policies. Feature policy will be a moving target and good reference docs will definitely be needed.

For now, there are a couple of ways to see what features are controllable.

- Check out our Feature Policy Kitchen Sink of demos. It has examples of each policy that's been implemented in Blink.
- Check Chrome's source for the list of feature names.
- If you have the `--enable-experimental-web-platform-features` flag turned on in `chrome:flags`, query `document.policy.allowedFeatures()` on `about:blank` to find the list:

```
["geolocation",
 "midi",
 "camera",
 "usb",
 "magnetometer",
 "fullscreen",
 "animations",
 "payment",
 "picture-in-picture",
 "accelerometer",
 "vr",
...
```

- Check chromestatus.com for the policies that have been implemented or are being considered in Blink.

To determine *how* to use some of these policies, check out the spec's GitHub repo. It contains a few explainers on some of the policies.

FAQ

When do I use Feature Policy?

All policies are opt-in, so use Feature Policy when/where it makes sense. For example, if your app is an image gallery, the `maximum-downscaling-image` policy would help you avoid sending gigantic images to mobile viewports.

Other policies like `document-write` and `sync-xhr` should be used with more care. Turning them on could break third-party content like ads. On the other hand, **Feature Policy can be a gut check** to make sure your pages never uses these terrible APIs!

Pro tip: Enable Feature Policy on your own content before enabling it on third-party content.

Do I use Feature Policy in development or production?

Both. We recommend turning policies on during development and keeping the policies active while in production. Turning policies on during development can help you start off on the right track. It'll help you catch any unexpected regressions before they happen. Keep policies turned on in production to guarantee a certain UX for users.

Is there a way to report policy violations to my server?

A Reporting API is in the works! Similar to how sites can opt-in to receiving reports about CSP violations or deprecations, you'll be able to receive reports about feature policy violations in the wild.

What are the inheritance rules for iframe content?

Scripts (either first or third-party) inherit the policy of their browsing context. That means that top-level scripts inherit the main document's policies.

`iframe`s inherit the policies of their parent page. If the **`iframe`** has an **`allow`** attribute, the stricter policy between the parent page and the **`allow`** list, wins. For more information on **`iframe`** usage, see the `allow` attribute on iframes.

Disabling a feature policy is a one-way toggle. Once a policy is disabled, it cannot be re-enabled by any frame or descendant.

If I apply a policy, does it last across page navigations?

No. The lifetime of a policy is for a single page navigation response. If the user navigates to a new page, the `Feature-Policy` header must be explicitly sent in the new response for the policy to apply.

What browsers support Feature Policy?

See caniuse.com for the latest details on browser support.

As of now, Chrome is the only browser to support feature policy. However, since the entire API surface is opt-in or feature-detectable, **Feature Policy lends itself nicely to progressive enhancement**.

Conclusion

Feature Policy can help provide a well-lit path towards better UX and good performance. It's especially handy when developing or maintaining an app since it can help avoid potential footguns before they sneak into your codebase.

Additional resources:

- Feature Policy Explainer
- Feature Policy spec
- Kitchen Sink Demos
- Feature Policy DevTools Extension - Tester for trying out feature policies on a page.
- chromestatus.com entries

Last updated July 20, 2018.