Public

Deprecating the chrome and resource protocols

Dave Townsend, June 2023 & Gijs Kruitbosch, December 2024

(original: Deprecating the chrome and resource protocols)

Summary

The `chrome` and `resource` protocols are used to reference internal resources that ship with Firefox such as JavaScript code, UI definitions and styles. They are an unnecessary complexity that make it harder for developers to understand the application's structure and for automated tools to identify potential errors. This document proposes an alternative that we can move to and which solves these problems. **This plan intentionally does not include completely removing the `chrome` and `resource` protocols instead just covering the bulk of uses.** After that we will have a clearer picture of whether we want to invest additional resources in completely removing them.

The Problem

Large parts of the front-end of Firefox including JavaScript, HTML and CSS code ship as files in an archive in the application. Internally they are referenced via URLs of a similar form to the http/https URLs used on the web but instead use the `chrome` or `resource` protocol. For example the main application UI is referenced as `chrome://browser/content/browser.xhtml` while one of the JavaScript modules that handles application update is at `resource://gre/modules/UpdateService.sys.mjs`. These protocols are substituting protocols, that is the path to the file in the URL differs from the path to the file in the source tree. While technically possible, it is non-trivial to determine where in the source tree a given `chrome` or `resource` url actually refers to. For example, the update service JavaScript module actually lives at `toolkit/mozapps/update/UpdateService.sys.mjs`.

This discontinuity between the path to the file in the running application and the path in the source tree makes it harder for developers to find the code they are looking for. When one script references another the developer has to do an additional translation step to actually find the referenced code. Similarly when a developer knows what script from the source tree they want to use they again have to translate that path to the correct internal URL.

Developers have a reasonably reliable means to do this translation, we have very few filename collisions and so searchfox can show the location in the source and example uses for existing cases. The developer can use context to resolve any collisions. This is still a manual process however. The problem is harder for automated tools. Our JavaScript linting tools support resolving imported code to verify that the referenced scripts exist and export what we try to import but these checks are currently disabled as we have no automated way to perform the translation. Code editors can also provide helpful intellisense support on imported symbols but only if they can map the import path to the source file.

The Proposed Solution

We propose implementing a new URL protocol (obligatory xkcd) to replace `chrome` and `resource` for many cases. Instead of being a mapped protocol, the path used in the URL will exactly match the path to the file in the source tree. So for our update service example the URL used would be `moz-src:///toolkit/mozapps/update/UpdateService.sys.mjs`. This makes it trivial for developers and automated tools to translate between in-product URLs and source tree paths. The protocol name could be anything but `moz-src` is a logical choice. Some files contain content generated at build time either via the preprocessor or other custom build scripts. These would be referenced at `moz-src://generated/<path>` where the path is the path in the source tree to the file's template. This still gives developers an indication of where to look for the source code but indicates that the content is modified somehow. It also allows us to flag `moz-src://generated/*` as references that shouldn't be subject to some checks by automated tools.

Currently various path translations are used to decide where in the build directory relevant files are stored. All files to be referenced by this protocol would instead be stored in the omnijar under `moz-src/<path-to-source-file>` for source files and `moz-src-generated/<path-in-objdir>` for generated files.

With these changes and some configuration to teach ESLint and VS Code about the new protocol we can see immediate benefits from ESLint warning about unknown imports and VS Code being able to context click through import statements.

While in some cases files in the resource and chrome protocols can be made accessible to web content this would not be possible for the `moz-src` protocol. For the time being we would continue to use the existing protocols for that case. A future proposal could suggest an alternative.

Implementation Plan

✅Stage 0 - Implement the `moz-src` protocol handler

The protocol handler itself is straightforward to implement as it behaves in a very similar fashion to the existing resource protocol handler. We already have a prototype implementation that appears to mostly work.

✅Stage 0.1 - Ensure the `moz-src` protocol handler works with fastload/xulcache

Without this the move is likely to incur a performance penalty. We should get ahead of this so that large commits moving all the files (which are hard to revert) don't have us time-pressured to figure this out.

Stage 1 - JS Modules and Actors

✅JavaScript modules and actors are currently just copied to the appropriate place in the build directory and then referred to via existing substitutions in the resource protocol handler. It is straightforward to instead copy them to the location targeted by the `moz-src` protocol. We have a prototype implementation that adjusts the build system to do this for modules, actors can be handled similarly. Straightforward scripts can be used to automatically replace existing in-tree references to the modules.

❌Note that Firefox on iOS re-uses some of Firefox's JavaScript modules (e.g.) and in their code bundling replaces the `resource` URIs with the correct paths. This would have to be adjusted to take account of the new protocol, but otherwise will continue to work.

✅Note also that there is a small number of sys.mjs modules where we rely on the resource mappings to provide a different implementation in different applications, see e.g. the Android implementation of BrowserUsageTelemetry vs the Desktop one. This is an attempt at a complete list based on multiple copies of modules with the same file name existing:

- BrowserUsageTelemetry.sys.mjs
- ExtensionBrowsingData.sys.mjs - android and desktop
- PromptCollection.sys.mjs - android and desktop - but these get packaged at different paths and are relying on XPCOM to obtain the service, not the module path.
- SessionStoreFunctions.sys.mjs - android and desktop - but these get packaged at different paths and are relying on XPCOM to obtain the service, not the module path.
- FormAutofillPrompter.sys.mjs - android and desktop but sourced via conditional JAR packaging and a custom `resource` prefix (see below) so probably OK for this stage?
- FormAutofillStorage.sys.mjs (see previous item)
- AboutCompat.sys.mjs - seems both android and desktop have copies of the web compat extension. This is probably fine?

✅Effectively this is using `resource` URIs as a dependency injection mechanism. Ideally we should probably evaluate an alternative. It looks like the solution for SessionStoreFunctions and PromptCollection (where C++ consumers simply `do_getService` and the mapping is provided via the moz.build file's `esModule` property, without the compiled code having to rely on a `resource` URI at all, only the contract ID and interface) could be used for the other cases. In the short term, we can continue accessing these files using `resource:` URIs and packaging them in the previous/existing locations by using `FINAL_TARGET_FILES.modules` (or similar) instead of `EXTRA_JS_MODULES`.

✅bug to update BrowserUsageTelemetry and ExtensionBrowsingData

✅ https://bugzilla.mozilla.org/show_bug.cgi?id=1951644

✅https://bugzilla.mozilla.org/show_bug.cgi?id=1950727 - Note also that Background tasks rely on `resource` registration right now. This may need to change, or we may keep using `resource` for these modules for now.

✅Note also that the about:sync add-on relies on sync module `resource` URIs and that and other add-ons in the mozilla-extensions org will need updating.

~~Stage 1.5 - Optionally allow relative references via ChromeUtils~~

~~Currently when lazily loading JavaScript modules via ChromeUtils relative URLs do not work. One small downside of the new protocol is that URLs tend to be longer. In some cases we can reduce the length by using a relative URL so we may choose to fix this issue in ChromeUtils. ES module import syntax already supports relative URLs.~~

✅Stage 2 - Import Resolvers for ESLint and VS Code

We have a prototype implementation of import resolvers for ESLint and VS Code that allows them to understand the `moz-src` protocol.

❌Stage 3 - JS Module Special Cases

There are a few cases where JS modules are being referenced with custom resource protocol substitutions. For example `resource://services-common/kinto-http-client.js` and `resource://featuregates/FeatureGate.sys.mjs`. There may need to be some manual work to move these to the `moz-src` protocol.

There are also some uses of `resource` from “extension” code (e.g. `formautofill`). Nominally we want to be able to dynamically replace this code for users by updating the builtin extension out-of-band. This will need more thinking; for the work in this document we do not intend to touch this.

❌Stage 4 - move chrome content files

Files currently referenced under `chrome://<ns>/content/*` can be moved to the `moz-src` protocol. The existing `jar.mn` files that control these files would need to be converted into entries in `moz.build` files however this can likely be done in an automated fashion for 95% of the cases. Updating in-tree references should also be scriptable.

This is split up because there are more interactions with these files than with modules, and they vary based on the type of file. The following chunks are parallelizable and/or splittable, and are ordered here purely by ease of reading (*not* complexity! So we may not want to start with 1.)

For all things other than toplevel pages, we will likely need to adjust CSPs so the resources continue to load (ie add `moz-src` in addition to `chrome` to CSPs of about: pages and similar documents).

1 - Move chrome .xhtml/.html (toplevel pages)

More migration is needed:

1. We'll need to migrate/map XUL store entries (likely in BrowserGlue's `_migrateUI`)
2. There is more docshell/windowing code that relies on these being `chrome://` URIs that will complain once we swap this (that won't complain for e.g. image or script loads)

2 - Move files that are overridden

There are surprisingly few of these inside `mozilla-central`, but a few more in comm-central. **That includes overrides of browser files with Thunderbird or related files**. We will need to work with the Thunderbird team to keep this mechanism working in some way. This may mean keeping them on `chrome://` for now.

At the time of writing, the list of comm-central overridden mozilla-central files is about 65 lines long. The vast majority are icons. The icons could likely be fixed in Thunderbird with custom CSS. Other files are:

- commonDialog.xhtml (alert/prompt/confirm etc.)
- utilityOverlay.js (loaded directly by devtools so comm-central has had to create a copy. We should fix devtools.)
- (more here)

3 - Chrome stylesheets

Stylesheets represent a harder problem as in many cases different source files are used for different platforms. For instance `chrome://browser/skin/browser.css` references `browser/themes/osx/browser.css` on macOS but `browser/themes/windows/browser.css` on Windows. For these cases the proposal is to either use multiple `<link>` elements in the source document with the `-moz-platform` media query to select which is actually loaded or define a shared stylesheet with common styles for all platforms and in that use `@import` syntax with the media query to include platform specific styles where needed.

It would likely make sense to do the work to change the loading mechanism while continuing to use `chrome://` URIs, and then swap over the protocol as a second step. This would derisk things and make it easier to trace regressions.

4 - Move images

Images themselves are straightforward to move but there are likely imglib optimizations/conditions (e.g. for `-moz-context-properties` or things like caching) that are conditional on images being loaded over `chrome://` that may need adjusting.

5 - Move other chrome content files

This includes script and JSON files, and potentially any other filetypes used via `fetch` or similar. Given that we moved modules in step (2) this *should* be reasonably hitch-free.

Stage 5 - consider moving other `resource` URIs (e.g. read via fetch/XHR)

In principle this should be straightforward, though it may not be worth automating as there are likely not many of them.

Given web access requirements, this won't be possible for e.g. the UA sheets.

Open Questions

(crossed out means these have been integrated in the respective steps above where they are best addressed)

~~Xulstore uses `chrome` urls to store the persisted DOM settings. If we change the urls used we may need to perform some kind of migration step.~~

There are some dynamic resource://testing-common registrations in our test harness that aren't easily accommodated here. These will remain as resource registrations for the time being until we decide whether to do anything with them.

~~Firefox on iOS re-uses some of Firefox's JavaScript modules (e.g.) and in their code bundling replaces the `resource` URIs with the correct paths. This would have to be adjusted to take account of the new protocol.~~

~~The `-moz-context-properties` CSS rule may need updating to work with the new protocol.~~

Localization files (.ftl, .properties) use `chrome` and `resource` URIs, which need to be mapped to source locations matching the current locale. Will these also be able to use the `moz-src` protocol?

~~We need to verify that Thunderbird isn't attempting to use chrome overrides for Gecko files.~~

~~Thunderbird add-ons are privileged and so unless we perform complicated mapping to provide backwards compatibility this could break many add-ons. There are additionally Mozilla privileged add-ons in the same boat but presumably we have control to update those. Some of these are equally impacted by the ESM modules work.~~ Thunderbird add-ons appear to use webextensions now.