MPArch: summary of //content/public changes

altimin@chromium.org
Last update: January 2021

This document provides a short summary of MPArch-related changes around //content's public interfaces. This document assumes that the reader is familiar with //content API and presents only a brief summary — this doc[1] will eventually have more details and explanations.

Glossary

Document — HTML document, building block of the web. Can be either a main document or embedded in an </iframe>. blink::Document in the renderer. No direct counterpart in the browser yet (content::RenderFrameHost is moving in that direction).
Page — a given main document together with all of its subframes (note: blink::Page is not a page in our definition as it's tied to a single process and sometimes doesn't change when the main document changes).
Frame — a container for documents e.g., <iframe>. Different from *document* as a frame can navigate from one document to another.

Changes resulting from MPArch

Stronger separation between page and tab concepts

"Page" and "document" concepts have been separated as a part of Site Isolation project and introduction of RenderFrameHost. Now we need to separate the "tab" and "page" concept, as some features used them interchangeably.

In particular, this involves:

- *Plumbing RenderFrameHost* instead of WebContents* where needed.*
  In many places code calls WebContents::FromRenderFrameHost only to call WebContents::GetMainFrame on the result a few layers away. Instead, RFH should be plumbed and RenderFrameHost::GetMainFrame should be called instead to stay within the same page. Most of the changes here are going to happen in embedders, but in some cases content/public boundary is going to be affected.

  This also includes cases where WebContents* at the moment is implied, in particular document/page-related WebContentsObserver callbacks.
- *Move document/page-related methods from WebContents to RenderFrameHost.*

[1] The linked doc is in process of being refreshed as of 15/01/21 — the overall plans are the same, but some recent changes in terminology haven't been updated yet, in particular the linked doc uses "PageHolder", while recent docs have been referring to the same concept as "FrameTree".

- *Storing per-document/page data in RenderDocumentHostUserData/PageUserData instead of WebContentsUserData*
  As multiple main documents can now exist simultaneously in the given WebContents, data associated with them should not be 1:1 with WebContents. In particular, WebContentsUserData should be only used for the states which are associated with the entire tab.

Many things can be implemented as either document/page-related or tab-related and the right choice depends on the circumstances. The main guideline for determining whether something should be document/page-related or tab-related is to check who is using it — if the feature has code running in the renderer[2], it's document/page-related and the methods and callbacks it uses should be document/page-related. If not, then the methods and callbacks are tab-related and their implementation should account for having multiple pages transparently to the user.

Impact on embedders

Embedders need to proactively audit their codebase before enabling MPArch-based features in their project — as the old APIs will continue to work, now-invalid assumptions are likely to cause issues of various scales and embedders will have to update the affected code to avoid these bugs (for example, by migrating on top of document lifecycle APIs, more on that below). From the Chromium side, we'll provide appropriate guidance by writing documentation and comments for affected APIs (e.g. WebContentsObserver, and in particular DidFinishNavigation).

In particular, embedders are expected to use the provided RenderFrameHost and plumb it around it as appropriate and avoid calling WebContents::FromRenderFrameHost. The exceptions are cases where the embedder wants to update a tab-related state on behalf of a document (e.g. show UI). In this case, the embedder should also check that the target document's lifecycle state allows it to be user-visible (see below).

More navigation types

Currently three new types of navigation are planned, which move an existing main document into a given frame: back-forward cache restore, prerendering activation and portals activation.

These will be exposed as methods on NavigationHandle (IsServedFromBackForwardCache is already present, IsPrerenderingActivation and IsPortalActivation are the working names. We are also considering an enum covering all five navigation types ("new document load", "same-document navigation" and the three aforementioned ones).

---

[2] Coming up with short and sufficiently comprehensive definition for tab/page separation is tricky given how many different cases have to be considered here — this is the best one-sentence approximation. The current attempt to describe this in more detail is here and we will iterate on it while looking into and fixing more features.

Also for portals and prerendering we are going to create dedicated frame trees, which means that the embedders will need a way to figure out if the “load new document” navigation is happening in the tab’s own main frame (aka *top-level frame*) or a different main frame (e.g. portal or prerender). Current (rough) plan: add NavigationHandle::IsInTopLevelFrame to complement NavigationHandle::IsInMainFrame.

Impact on embedders

Embedders should either start using a new document lifecycle API (more on that below) or check for the navigation type correctly. It is expected that most of the features can stop listening to navigations and start listen to more specific and simpler signals instead (e.g. document lifecycle notifications for document/page-level features or per-tab notifications for tab-level features) — in particular, the number of overrides for WebContentsObserver::DidFinishNavigation is expected to be at least halved.

Features which want to continue to listen to navigations (more complex features, like the ones that would want to modify navigation via NavigatonThrottle) would have to implement appropriate checks. In particular, places checking IsSameDocument now are expected to either start checking GetType() == kNewDocumentLoad or GetType() != kIsSameDocument instead, and NavigationHandle::IsInMainFrame() calls should be audited and replaced with in IsInTopLevelFrame() wherever needed.

Document lifecycle API

Given that listening to navigations is an already complex business which will become even more complex, we want to reduce the number of places which listen to navigations. In particular, places which listen to navigations to infer whether things related to a document should be cleaned up, and should be better served by document-related APIs instead.

The plan here is to add RenderFrameHost::GetLifecycleState and WebContentsObserver::OnRenderFrameHostStateChanged(RenderFrameHost* host), which should allow us to eliminate at least half of existing DidFinishNavigation usages [example].

Impact on embedders

Embedders should use document lifecycle APIs instead of observing navigation directly when possible, which should simplify their code.

Embedders should also check that the document is allowed to update tab-level state (incl. showing UI) before doing so (at the moment: RenderFrameHost::IsCurrent(), final name TBD).

Nesting

Existing methods on WebContents to traverse nested WebContents will be replaced with similar methods on RenderFrameHost, which will work for both nested pages and nested WebContents.

For example, WebContents::GetOuterWebContents will be replaced by RenderFrameHost::GetOuterFrame (or Page::GetOuterPage, details TBD). This should allow the existing code for features like find-in-page to work both with nested WebContents and nested pages.

Impact on embedders

Embedders would have to update the traversal code to use the new APIs, as the old APIs would be removed – problematic places would be easily identified by compilation failures.

content::Page

As page is 1:1 with the main document, all per-page state can be tied to the main document (after RenderDocument project is implemented and every navigation loading a new document creates a new RenderFrameHost, it will become 1:1 with the main RenderFrameHost). However, we're planning to separate per-page state into a separate interface for clarity and type safety, including ensuring that we don't have methods on RenderFrameHost, which can't be called on subframes and per-page state being accidentally associated with a subframe.

content::Page should be responsible for:

- Owning LocalMainFrame mojo interface
- Storing and providing access to per-page state, including arbitrary embedder state via PageUserData
- Hosting methods main-document-only methods [example].
- (probably) Supporting iteration over documents to replace WebContents::GetAllFrames/SendToAllFrames.

Impact on embedders

Embedders should use PageUserData for the data which should be associated with a given page — this is expected to be the migration path for some of the issues mentioned above in "Stronger separation between page and tab concepts".

Navigation Throttles

The NavigationThrottle interface at the moment is strongly tied with the different stages of network request creation and handling — the thing that the new navigations (bfcache restores and prerenderer/portal activations) do not do. In the short term, we'll emulate a fake network request. In the longer term, we'll think about how to evolve NavigationThrottle API to better reflect navigations which do not hit the network.

Impact on embedders

(note: the plans for NavigationThrottles are not finalised and it's not known whether we will keep the interface mostly as-is or will substantially evolve it).

If the interface will remain mostly the same: embedders will have proactively to audit their NavigationThrottles before enabling features which create new navigation types and ensure that their navigation throttles behave correctly for these navigation types (e.g. that there will be two sets of throttles running both for document load and a subsequent restore of the said document from bfcache).

In case of major API changes: embedders would have to migrate on top of the new API.

Overall impact on non-Chrome Chromium embedders

Chromium is gaining new powerful features (bfcache, prerendering, portals) that will require a set of changes to the embedders to guarantee safety and correctness as described above.

The architecture powering these features (MPArch) is going to incrementally evolve the //content/public API surface and not going to deprecate or remove many existing APIs, due to being closely aligned with existing navigation stack. This makes it easy for the embedder to rebase on top of new changes as MPArch develops, but also it makes it easy to miss the moment when MPArch-backed features will get enabled and when assumptions behind many usages of //content/ APIs become invalid.

When (and before) the MPArch-backed features will be enabled by default in Chromium, we would recommend embedders disable these features in their codebases if they are not ready and will strongly encourage them to update their codebases and enable these features (albeit embedder remains in control of this timeline). We will support and provide guidance to embedders (e.g. by updating the comments, writing docs and guidelines, and proactively communicating) based on our experience with the same migration for //chrome and //components.