Multiple Page Architecture [public]

{altimin, carlscab, lfg, haraken}@chromium.org
Last update: 2020 Jan 15
Status: PUBLIC

This document is intended to concisely describe a high-level plan of ***Multiple Page Architecture***. For technical details, please read summary of //content/public API changes , core navigation changes and feature-side changes.

This doc (Google-internal) summarizes multiple discussions between MPArch and CSA teams and contains links to further documents.

[Note: In this document, a "page" refers to a tree of documents with a given root, and a “page” is 1:1 with “main document”, which matches intuitive definition of “web page” as in “type URL into omnibox and load a new web page”]

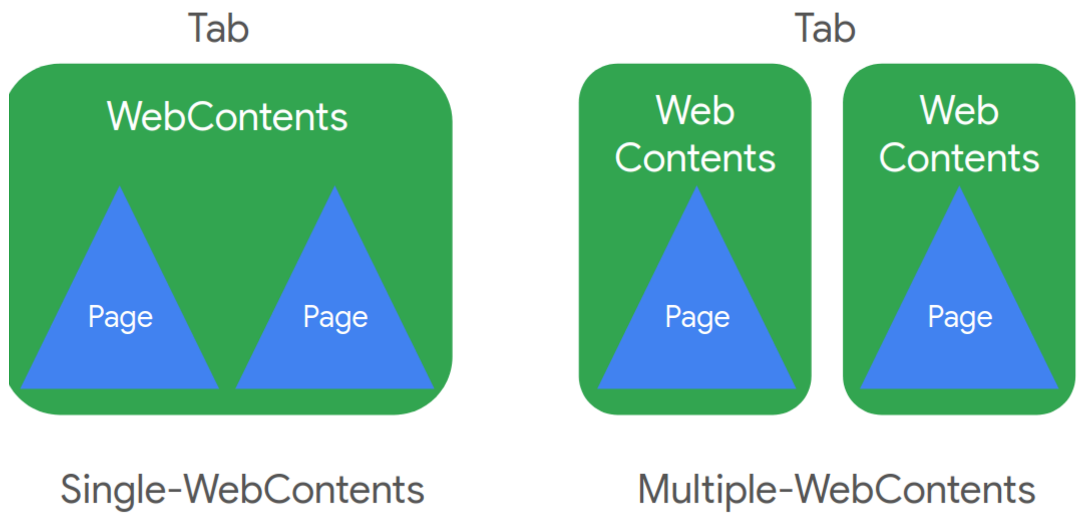
Approvals

Role	LDAP	LGTM / date
Eng owner, //content/ owner	creis@	LGTM, 1/14/2021
Eng owner, //content/ owner	darin@	LGTM 1/21/2021
Eng owner, //content/ owner	jam@	LGTM 1/21/2021
//content/ owner	kinuko@	lgtm on Oct 19, 2020 but conditioned that concerns/questions from CSA team are all resolved)
//content/ owner	nasko@	LGTM 2/16/2021
//content/ owner	alexmos@	LGTM 1/27/2021
<add your role>	<add your name>	

Why we need it

Recently it turned out that back/forward-cache, portals, guest views and pending deletion documents face the same fundamental problem of having to support multiple pages in the same tab, but are implemented with different architecture models. bfcache and pending deletion documents are implemented with a Single-WebContents model, whereas Portals and Guest

Views are implemented with a Multiple-WebContents model. After extensive discussions between Chrome Security Architecture team, the BFCache team, the Blink Architecture team, the Portals team and others, we concluded that bfcache and Portals should converge to a single shared architecture (see the "Alternative Considered" section). This is crucial to minimize the complexity of writing browser features and on top of and within and minimize the maintenance cost across the codebase.



In addition to bfcache and portals, prerendering, fenced frames, guest views and pending deletion frames are facing the same problem: **how to support multiple pages in a given tab.**

Feature	Requirements			
	Multiple pages within a tab	Multiple active pages within a tab	Nested pages	Navigating to an existing page
pending deletion *documents in process of deletion, which wait for renderer to run unload() handlers*	Yes	No	No	No
back/forward cache *cache for the navigated-away-from pages to instantly navigate back to them*	Yes	No	No	Yes
prendering *loading the page in advance before user navigates to it*	Yes	Yes, but with a limited set of features	No	Yes
fenced frames *isolated third-party frames with first-party storage access*	Yes	Yes	Yes	No

portals *embedded pages which can be activated to seamlessly become a new active page*	Yes	Yes	Yes	Yes
guest views *embedded infra backing <webview> tag implementation*	Yes	Yes	Yes	No

If we don't invest in the architecture:

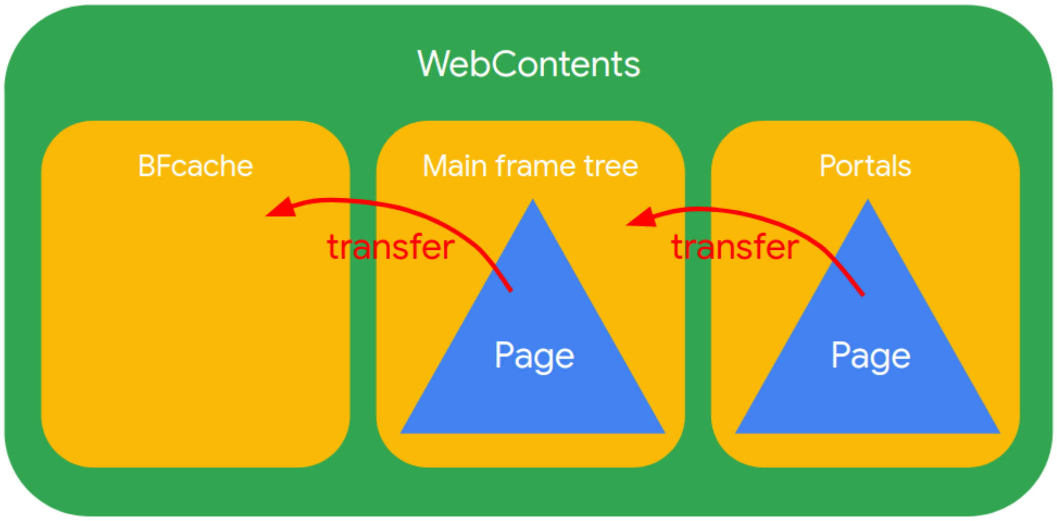
- **The six use cases need to solve the same problems individually, which will lead to browser features having to deal with multiple incoherent cases**, greatly increasing the complexity across the codebase.
- All WebContentsObservers need to be written to support both Single-WebContents and Multiple-WebContents models. **This is strictly worse than unifying the architecture into one model**.

From the perspective of technical debt / engineering velocity, there's no option of not investing in the unified architecture. Investing in the unified architecture also minimizes the time to get prerendering, portals and fenced frames to the shippable state. The tradeoff is that some assumptions made by existing Chrome and embedder code no longer hold, and thus some existing code needs to be updated to handle the possibility of having multiple pages per WebContents.

Goals

The goal of Multiple Page Architecture is to **enable projects** including back/forward cache**,** portals, prerendering, fenced frames in a way that **minimizes the complexity** exposed to the browser features.

Specifically, Multiple Page Architecture:
- enables one WebContents to host multiple (active / inactive, visible / invisible) pages
- allows navigating WebContents to an existing page



For example, when a page moves to bfcache, the ownership of the page is transferred from the main frame tree to bfcache. When a portal is activated, the ownership of the page is transferred to the main frame tree.

This requires to **clearly split the tab and page / main-document concepts.** This is closely related to and greatly benefits from Site Isolation's effort to split page and document concepts.

At the moment, many WebContents APIs and //content embedders assume that there is only one page in the tab and implicitly attribute events and signals to the active main document; the most prominent example here is embedders storing per-page state in WebContentsUserData and resetting it in WebContentsObserver::DidFinishNavigation. This will no longer work, as the old document / page is not necessarily destroyed after DidFinishNavigation and the new document / page is not necessarily newly created. Thus, we propose an audit and proactive communications to help update existing and future code to meet the new invariants.

Design overview

This section provides a brief high-level design description. If you have any comments or questions, please add comments to this doc for //content/public APIs, this doc for the core navigation work, and this doc for feature-side work.

Core navigation work

Core navigation work will 1) provide the ability to host multiple pages in the same WebContents, 2) allow one page to embed another and 3) allow WebContents to navigate to existing pages.

In particular, this includes:

- Allowing multiple FrameTrees to be created in the given WebContents in addition to the existing main one. (For example, portals and prerendering would create additional FrameTrees.)
- Adding a new navigation type, which navigates a given FrameTree to an existing page. (For example, bfcache will use it for history navigation restoring the page from the cache. Prerendering will use it to instantly navigate to a prerendered page.)
- Modifying RenderFrameHost to avoid knowing its current FrameTreeNode/FrameTree to allow the main RenderFrameHost to be transferred between different FrameTreeNodes.

Feature-side work

Feature side work **fixes existing features** relying on the now-invalid assumptions (like one tab only having one page at any given moment). We evolve //content/public APIs to **simplify these fixes** and **provide guidance and documentation** to browser feature developers.

The main components include:

- Auditing the codebase to identify problematic places.
- Removing implicit attribution of events to the tab's current main document by plumbing explicit RenderFrameHost* / NavigationHandle* references in more places and eliminating WebContents::GetMainFrame (and equivalent) calls where appropriate.
- Replacing WebContentsUserData with RenderDocumentHostUserData and PageUserData for the data which is associated with a given document or a given page, respectively.
- Introducing document lifecycle APIs to reduce the need for features to listen to complex navigation events for simple tasks like hiding document-associated UI elements after the page is navigated away. This eliminates the need for the majority of DidFinishNavigation calls.
- Adding information about new navigation types to NavigationHandle for more powerful features (like NavigationThrottle) which want to observe different stages of a navigation.
- Adding new traversal APIs (like RenderFrameHost::GetParentOrOuterFrame), which work both for nested pages and nested WebContents. This allows a shared implementation for both embedding types during the transition period.

This is a large amount of work but they are highly parallelizable. We will also proactively communicate a guide for these changes to other Chrome team members and other Chromium embedders, to help meet the new invariants in existing and future code.

Roadmap

The amount of work needed to fix all of the features across the large codebase is massive, so we are planning to incrementally deliver impact by enabling MPArch-based projects one by one and limiting the initial set of supported platforms and staging work accordingly. As the majority of the expected impact of MPArch-based projects lies on Android, we will start by focusing only on Android (both Clank and WebLayer) and adding support for desktop-specific features (some of which like extensions are non-trivial to support). Note: we are not yet planning to do this for WebView, as MPArch is based on Site Isolation, which is not yet available on WebView.

From the project perspective, we start with back-forward cache on Android being implemented and will focus on enabling prerendering (both of these projects can evict pages at arbitrary moments in time, which makes it much easier to deal with the long tail of features). Then we'll enable fenced frames and portals on Android by fixing necessary features, after which we will bring MPArch to desktop.

Alternatives considered

The Single-WebContents model vs. Multiple-WebContents model was discussed extensively in this doc and this doc. Our assessment is that unifying into the Single-WebContents model will be a win. It may require a larger number of smaller fixes than the multiple-WebContents model, but the multiple-WebContents model would require much more complex fixes and more new APIs.

The main benefit of the single WebContents model is that it allows close alignment with existing navigation stack, which provides support for navigating given WebContents to a new page, which allows to reuse the existing logic for handling page swaps in a given tab after slight modifications (to account for the fact that the swapped-in page might already be loaded).(so the code needs just to be updated to learn that the new page is not necessarily being loaded from scratch but rather might already have some state, instead of adding new set of APIs).

The single WebContents model also avoids the main downside of the multiple WebContents model; i.e., the multiple WebContents model loses an intuitive 1:1 mapping between "tab" and WebContents. This makes it harder for browser features to implement tab-level concepts, which affects the majority of the features.

Risks

Detailed technical risks are described in the other three docs. A high-level risk is that the feature-side work has a super long tail and may increase the eng cost to unblock Portals, which needs to support all features on Android (i.e., there is no option of blocking features). We should audit the amount of required work carefully, and if it's too much, explore ways to unblock Portals without waiting for the full feature support.