PUBLIC

about:home Performance Analysis

by Mike Conley

Note: `about:home`, about:welcome and about:newtab point to the same underlying infrastructure. about:welcome is distinct in that it will often show additional information over top of that resource. For the purposes of brevity, all three of these pages will simply be referred to as "`about:home`".

Note that "Discovery Stream" is also used to describe a *particular mode* of about:home, and some users are already receiving this mode. We currently appear to support both modes, but there are in-progress plans to transition all users to the "Discovery Stream" mode.

Questions for the analysts

- *How does about:home work?*
- *Are there ways it could be made more efficient so that the perceived startup of the browser in the default configuration is improved?*
- *What can be done to protect this part of the codebase from unintentional performance regressions?*

Scope of analysis

I'm going to be looking at the following scenarios:

1. Pre-existing profile startup, no automated session restore - the common case
2. First startup - a brand new profile is being started
3. Post-upgrade startup - the browser is starting, having just completed an upgrade
4. about:home pages loaded in new windows
5. Preloaded about:newtab load

In this analysis, we assume that the privileged about content process is disabled, since that feature has been disabled for all channels.

Although this analysis mainly focuses on startup, I will dedicate a small chunk of time for analyzing how about:home might impact the responsiveness of the browser during normal usage.

PUBLIC

Analysis was done on revision b4755981c138. Note that sections of the about:home code is imported from GitHub, and the associated GitHub revision is fe58d5d.

Primers

A (very) quick primer on React and Redux

React is a popular front-end library that is used in Firefox for both about:home, our Developer Tools UI, and our profiler analysis tool.

The documentation on React is quite good, and can help familiar any readers with the library. For the purposes of this analysis, suffice it to say that React acts as a function that knows how to take some state object, and knows how to convert that state into DOM operations to construct a UI that is defined within various React components.

Redux is a separate library, but it often goes hand in hand with React. Redux makes it possible to create "stores", where each store is the source of truth for the state of some part of an application. Actions (specially-crafted JavaScript objects) are dispatched into a Redux store, and "reducers" are used to translate those actions into internal state updates. Actions are the *only way* to update the state.

When used with React, these internal state updates usually trigger a "render" in React, which causes the UI to reflect the new state.

So the pattern is:

1. An application has a Redux store which is the source of truth for application state
2. Actions are dispatched into the Redux store to update the internal state. These actions are often in response to user input events.
3. Reducers process those actions and update the internal state of the store
4. If the internal state is changed, React is told to render the new state

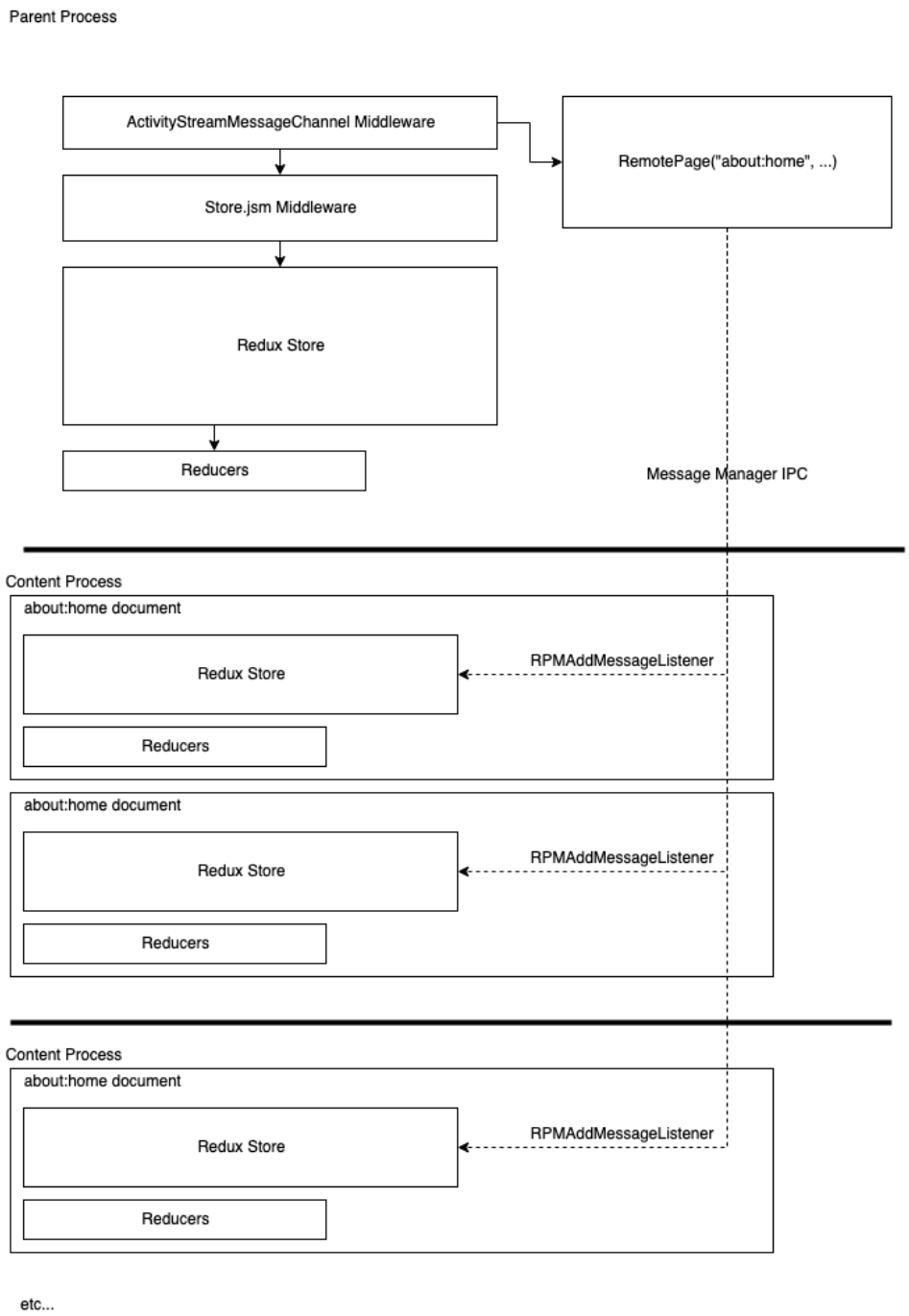
How about:home uses Redux stores

about:home's way of using Redux is unusual for a number of reasons.

The first reason is that a Redux store exists in the parent process, and copies of that store are sent down to each instance of about:home. After copying, each about:home instance then has an independent copy of the store which can be updated independently. Having each about:home instance have their own copy of the store makes sense since each about:home instance can be in a variety of states independent from one another (for example, one might be in the Top Sites editing state, whereas another might be in the default state).

The second reason is that custom middleware is set on the Redux stores in the parent and child processes so that the stores can also act as IPC message relays.

The diagram below tries to make this clearer:



The Redux store in the parent process is constructed such that the `ActivityStreamMessageChannel` middleware gets a chance to process each action before it reaches the parent process Reducers. Specially crafted actions can then be copied and dispatched over a RemotePageManager message port to one or more children.

Similarly, in each about:home instance, the Redux store has the capability to relay actions back up to the parent.

In both cases, actions need to opt-in in order to be copied like this to other processes. Actions opt in by having special properties set on them, and this is usually taken care of by helpers within the Actions.jsm file, for example, BroadcastToContent, AlsoToMain, OnlyToMain.

Note that in some cases, actions are relay-ed to another process, but are dropped before they reach the current process Redux store. For example, this action in an about:home document is only sent to the parent process, and never gets seen by the reducer functions for the about:home document.

As part of this analysis, each action type has been documented, including where they start from, where they're dispatched to, and whether or not they reach the reducer functions of either process.

A few words on Activity Stream vs Discovery Stream

Activity Stream was the original project name for the React-based about:home page. A more recent variation on this page was developed under the project name Discovery Stream.

From the user's perspective, Discovery Stream differs from Activity Stream in that Discovery Stream has more places for Pocket stories. Under the hood, Discovery Stream also has the capability of having its layout structure updated remotely. This capability was added so that the Pocket team (which now owns Discovery Stream) could quickly experiment with different layout variations. The experimentation has since concluded, and the layout is now hardcoded.

At this time, Discovery Stream is only shipping to a few locales, and we're likely to ship both Activity Stream and Discovery Stream for at least some of the foreseeable future, so any recommendations we make should take both into account.

Confusingly, there is also a component, `ActivityStream`, that is distinct from the "variation" of Activity Stream. `ActivityStream` is an essential component of how the `about:home` code initializes itself, and is used in both the Activity Stream and Discovery Stream variations.

PUBLIC

How about:home initializes

Disclaimer

This is a non-exhaustive walkthrough of a typical initialization of about:home. It doesn't capture *everything* that occurs, but I've tried to mention the *truly relevant* things that occur.

Also, unfortunately, there's non-determinism inherent in the startup timeline. I used this profile as my guide for laying out the description of execution in a realistic way.

Main entrypoint during pre-existing profile startup

Before any browser windows have been opened, the BrowserContentHandler asks the HomePage module for information on the initial URL to load, and passes that to the window arguments to the first browser window that it opens. In this scenario, this is `about:home`.

Within the scope of that first browser window, the `DOMContentLoaded` event handler for browser.xhtml fires, and `gBrowser.init` is called. This causes us to begin setting up the initial browser tab and browser for the window.

At this point, we create the initial browser with the appropriate content process type for `about:home`.

Within the load event handler of the initial browser window, the `browser-window-before-show` notification is fired, which SessionStore.jsm responds to. It causes SessionStore.jsm to queue a function to initialize the window for SessionStore, which will cause the `sessionstore-windows-restored` notification to fire. The point at which this function runs is non-deterministic, since it reads session state off of the disk asynchronously. For this scenario, I'll place the running of the `sessionstore-windows-restored notification` a few paragraphs below, as per the source profile.

Eventually, the initial browser paints its first frame, and the _delayedStartup method fires to perform post-paint work. It's during this time that `AboutNewTabService`:

1. Registers itself as an observer for a number of low frequency notifications, like `quit-application-granted`
2. Registers itself as a preference observer for some preferences
3. Sets some internal state flags
4. Calls `AboutNewTab.init`

`AboutNewTab.init` registers itself as an observer for some notifications, including `sessionstore-windows-restored`, and registers a `RemotePage` with the

`RemotePageManager` to manage all instances of `about:home`, `about:newtab` and `about:welcome` that are opened.

Back in `_delayedStartup`, we see if we know enough yet to load the initial browser's URI. Since we know we're loading `about:home`, we synchronously enter the `_callWithURIToLoad` callback, which causes us to call `loadOneOrMoreURIs` with "`about:home`".

After some indirection through tabbrowser.js and RemoteWebNavigation.jsm, we tell the `BrowsingContext` of the initial browser to load "`about:home`".

Inside the content process, the underlying `DocShell` knows to map `about:home` to a URL provided by a second copy of `AboutNewTabService` which is running in the content process. In this case, that URL is `resource://activity-stream/prerendered/activity-stream.html`.

Once the document global for this page is created, the process script for the content process is notified, and since a `RemotePage` was registered for `about:home`, a `ChildMessagePort` is constructed for it, which causes a `RemotePage:InitPort` message to be sent to the parent over the message manager. This is also where the `ChildMessagePort` adds a load event listener to the page so that it can send the `RemotePage:Load` event once `resource://activity-stream/prerendered/activity-stream.html` has finished loading.

At this point, `sessionstore-windows-restored` fires, and the observer in `AboutNewTab` fires. This queues up a function to run on the next tick of the event loop, which constructs an instance of the `ActivityStream` class.

We are now ready to instantiate `ActivityStream`.

Instantiating and initting `ActivityStream`

Upon instantiation, `ActivityStream` instantiates a `Store` from `Store.jsm`, which will be used as the master Redux state in the parent process. Importantly, instantiating the `Store` also causes an `ActivityStreamMessageChannel` to be instantiated (but not initialized), which will be used to wrap the `RemotePages` instance that will be used to communicate with the content processes hosting about:home documents[1].

After instantiation, `ActivityStream.init` is called. This reads some values out of the Preferences database to set some internal state, and then the `Store.init` function is called (and the `at.INIT` action is queued to dispatch on it once initting completes).

---

[1] This `RemotePages` instance will be instantiated during the initialization of the `ActivityStreamMessageChannel` later on during the `Store`'s `init` function.

Initting the parent process Redux `Store`

First, the `Store` initializes the `TelemetryFeed`, presuming that feed is enabled. The `TelemetryFeed` adds some observers and event listeners to existing windows.

The `Store` then awaits the instantiation of `ActivityStreamStorage`, which is an IndexedDB-backed storage bucket. By await-ing, we yield back to the parent process main thread event loop, and wait for the database connection to be set up[2].

Once the IndexedDB connection is opened, we resume the `Store's init` function, and instantiating instances of each feed class and storing them in a set that the `Store` owns.

Then the `ActivityStreamMessageChannel` is set up, **so from this point forwards, communications have been opened between the parent process and any about:home documents**.

Then the Store dispatches the `INIT` action on itself, which is handled by most of the registered feeds.

Initting the `ASRouterFeed` mainly instantiates and initializes the `ASRouter` component, which is mostly outside of this analysis, and will be ignored here.

Initting the following feeds is relatively cheap, and only involves adding observers, instantiating some small objects, or reading information from prefs:

- `AboutPreferences`
- `FaviconFeed`
- `NewTabInit`
- `PlacesFeed`
- `SystemTickFeed`

Initting the `PrefsFeed` involves reading a number of preferences from the preferences database, and then broadcasting them to each about:home document.

Initting the `SectionsFeeds` is similar, except that instead of sending prefs, a message is broadcast to each about:home document about each registered section state.

Upon initting the `TelemetryFeed`, it adds some observers and window listeners, and then might send PingCentre Telemetry on whether or not the user is configured to view the default about:home.

---

[2] There is no guarantee that this database connection will become available within a reasonable amount of time, which leads to the first major finding.

The next few initializations are complex enough to warrant their own sections.

Initting the `HighlightsFeed`

Upon initting the `HighlightsFeed`, the highlights are calculated and broadcast to all about:home documents. Calculating the highlights means calling into `NewTabUtils`' `getHighlights` method (via a `LinksCache`, which presumably acts as a timed cache on subsequent queries).

The first step in calculating the highlights is querying Places for recent bookmarks[3]. Then querying for recently Pocketed articles if Pocket is enabled, which means a network request to the Pocket servers looking for things that the user has Pocketed. Thirdly, Places is queried again for recent browsing history. A bunch of deduplicating and filtering pre-processing is done on the results to produce a results list from this information, and then we return back to the `HighlightsFeed` code.

The `HighlightsFeed` then queries a `DownloadManager` component for 1 recent download that succeeded, if one exists.

More sorting, de-duping and filtering work occurs with this list of results[4], and then the resulting list of highlights is broadcast to all about:home documents via the `SectionsManager`[5].

Back in the `HighlightsFeed` initialization function, a `DownloadManager` component is queried for recent downloads. This means adding a view to DownloadsCommon, which causes a maximum of 42 `onDownloadAdded` function calls to occur. These are debounced via a timer so that 1 second after the view is attached, a `DOWNLOAD_CHANGED` action is dispatched to the parent process Redux store, which causes the `HighlightsFeed` to recompute and broadcast to all children again.

Initting the `TopSitesFeed`

The TopSitesFeed sets some internal state, and then immediately starts to prepare a broadcast to all open `about:home` pages.

Preparing the broadcast starts by waiting for the `TippyTopProvider` to be initialized. The TippyTopProvider starts by accessing a .json file off of the main thread. Until this JSON file is read, parsed and returned, the `TopSitesFeed` is delayed from making its broadcast[6].

---

[3] For a maximum of 12 rows, which appears to be a constant limit set here.
[4] I'm only able to highlight a single line here, but the rest of the function is dominated by sorting, filtering and de-duping.
[5] There's some indirection via the `SectionsManager`, but the broadcast eventually occurs here.
[6] This is similar to this finding, and additional support to the argument that the default initial page should require as little computation as possible.

After the `TippyTopProvider` is initted, the top sites links are computed, which results in the initialization of the Search Service.

Then Places is queried for frecent pages, and each of those sites is checked against each engine in the Search Service to see if any of them should resolve to be search providers.

Blocked sites from the default top sites list are filtered out from the results, and then those too are checked against the Search Service to see if any of them should resolve to be search providers. Pinned links are then fetched - these are ultimately read out of preferences and parsed as JSON. Work is then done to determine if any other search engines should be injected into the Top Sites list. After some more sorting, combining and filtration, a graphical representation is computed for the top site - this might be a high-resolution favicon, a screenshot from the thumbnailer, or a custom image. Requests to fetch these images are sent in parallel.

Next, the IndexedDB database is queried for information about the Top Sites section. Finally, the set of links computed for the Top Sites are then broadcast to all about:home documents.

Interestingly, it looks like TopSitesFeed then recomputes any default search engines that should appear in the Top Sites section, and broadcasts them to all about:home documents.

Initting the `DiscoveryStreamFeed`

Upon construction, the `DiscoveryStreamFeed` instantiates a `PersistentCache`, which it uses to store state in a JSON file.

Upon initialization, the `DiscoveryStreamFeed` populates its internal configuration state by parsing some JSON out of the preferences database on the main thread. It then broadcasts this information down to each about:home document.

The next sections presume that Discovery Stream is enabled. If Discovery Stream is not enabled, they are skipped.

The `DiscoveryStreamFeed` reads the `PersistentCache` JSON file off of the disk (using a separate thread to read from the disk and decode the JSON). If there are any contents within that cache, then the age of the cache is then recorded in Telemetry.

The `DiscoveryStreamFeed` then broadcasts the hardcoded layout down to all about:home documents, and sometimes follows that up with a message updating every about:home document about the sponsored content network endpoint, and then another message for any placements for that sponsored content.

If the user is configured to show sponsored content and the sponsored content cache has expired, then a network request is opened up to the sponsored content endpoint to get a more recent set. The updated information is then written to the cache.

Simultaneously, if it is determined that DiscoveryStream is enabled, then a list of "content feeds" is put together to potentially retrieve updates on various parts of about:home that can be updated over the network. In practice, this is only the "CardView"[7], which to end-users is what presents the Pocket stories.

If a network request needs to be made to those content feeds (if, for example, the local cache doesn't exist or has expired), then the DiscoveryStreamFeed waits until the network request completes before updating all about:home documents with the new feed data, and then sending an additional message to tell those documents that loading of those feeds has completed.

At this point, another attempt is made to update the cached data for things like sponsored content and the Pocket stories in the background.

Initting the `TopStoriesFeed`

If Discovery Stream is not enabled, then upon instantiation, the `TopStoriesFeed` then instantiates a `PersistentCache`, which it uses to store state in a JSON file.

The `TopStoriesFeed` waits until the `SectionManager` has finished initializing itself, and then checks to see if Discovery Stream is enabled. If so, its initialization bails out immediately.

If not, then the JSON file is read off of the disk and parsed. If pre-existing stories and affinities have been calculated and stored for this user profile, these are used to construct an `AffinityProvider` that's presumably used to reorder the stores presented to the user.

If it turns out we don't have any cached Pocket stories to display, some are downloaded and parsed off of the main thread and then run through the transformation that applies any pre-existing affinities. These stories are then cached to the JSON file.

Similarly, if we don't have any cached Pocket Topics[8] to display, some are downloaded, parsed off of the main thread, and then cached to the JSON file.

At this point, a message is sent to every `about:home` document letting them know about the stories and topics they should be displaying.

---

[7] This is because the "CardView" is the only component of the layout that offers a `feed` property.
[8] A Pocket Topic is a category for a story. These are typically displayed beneath the Pocket stories in `about:home`.

The first `about:home` retrieves state from the parent process to render with React

While all of the above is happening, the about:home document is loaded in a content process. This document is assembled at build time and contains a great deal of code. For simplicity in reading, instead of referencing the generated code, I'll be referencing the JSX and JavaScript that is used to generate the final document.

After loading all of the required libraries, the about:home document initializes its local Redux store. This is a store that is modified with middleware that knows how to communicate back and forth with the parent. A `DetectUserSessionStart` class then uses that store to tell the parent that the initial about:home is visible.

The about:home document then sends a `NEW_TAB_STATE_REQUEST` action to the parent. This message requests that the parent sends down a copy of its Redux state down to the content for it to render and maintain separately.

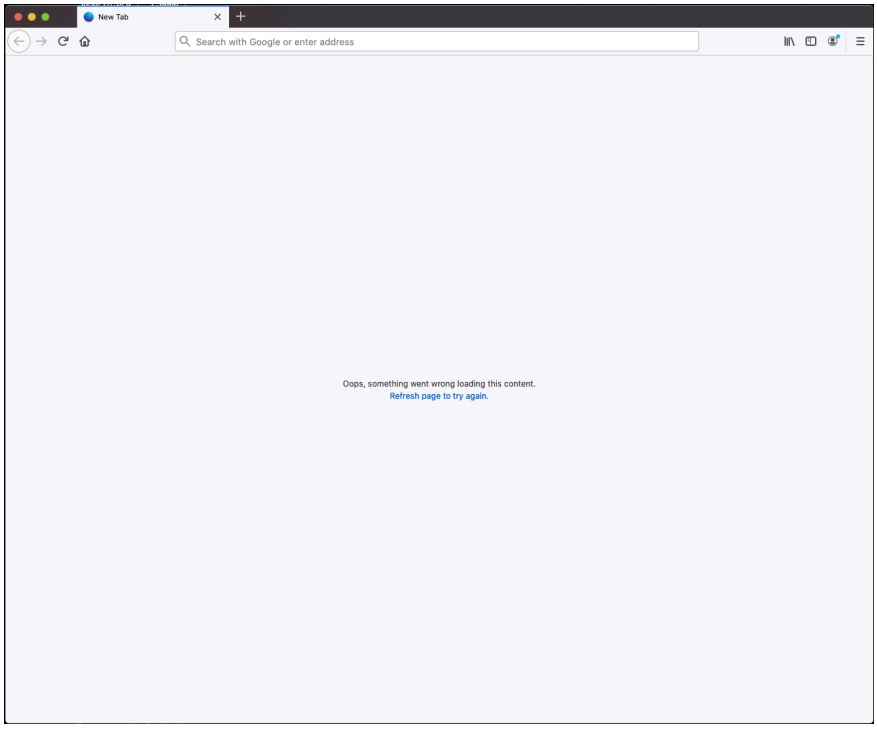
While it waits for a response to that message, the empty store is rendered by React, but this results in nothing meaningful displayed to the user.

At this point, it's possible that a number of other messages sent by the parent during startup will be received and processed by the content process store before the parent sends the Redux state down to the child. Those messages, however, will be ignored. The only non-hydration message that the store will listen for initially is the `INIT` message, which will cause the store to re-request the Redux state (this handles the case where the about:home document finishes initializing before the parent does).

Eventually, the parent receives the `NEW_TAB_STATE_REQUEST` action, and then replies to that initial about:home document with a copy of the Redux state via a NEW_TAB_INITIAL_STATE action.

Unfortunately, the first batch of state that the parent sends down does not cause anything to render. If left to its own devices, it'll eventually render this error page:

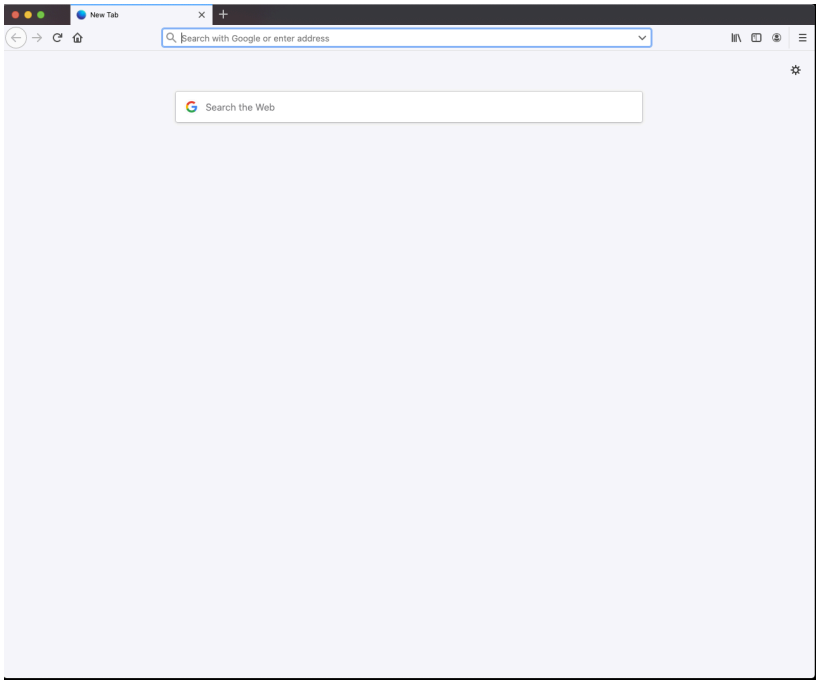
PUBLIC



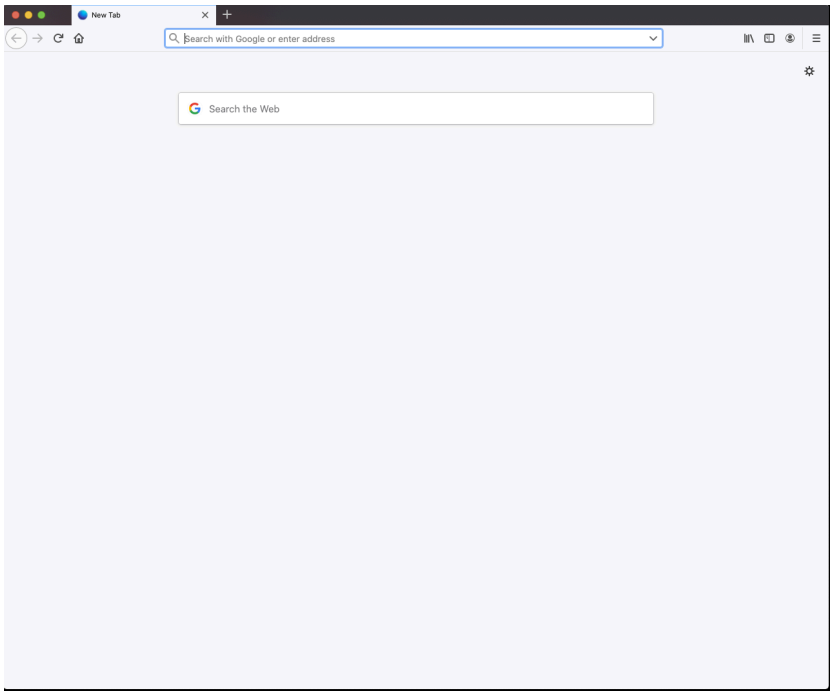
For the single about:home load on startup (having disabled the preloaded about:newtab), the following messages are received after the initial hydration request:

PUBLIC

SECTION_REGISTER

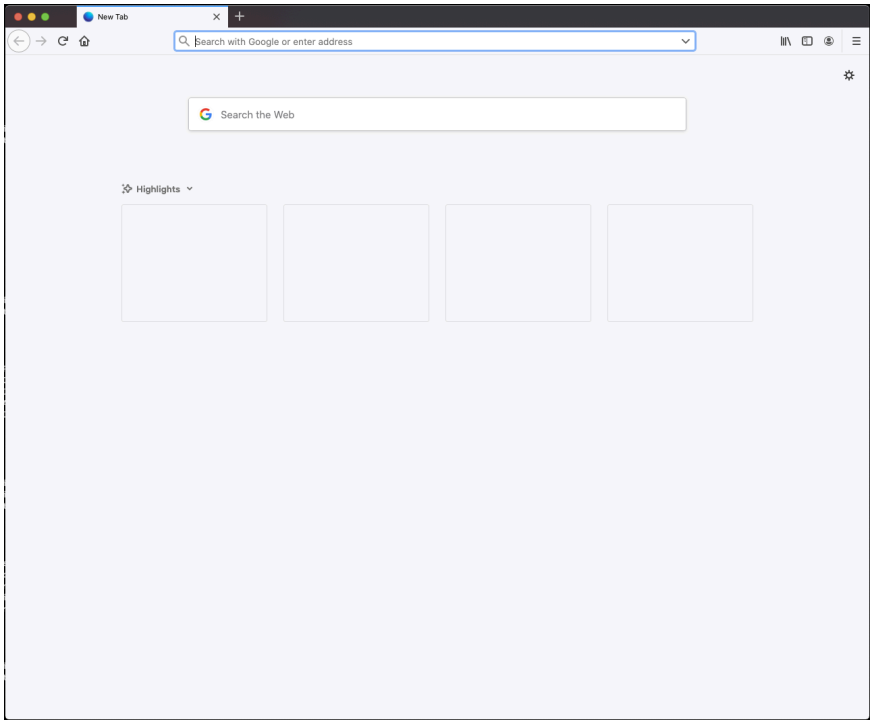


SECTION_REGISTER

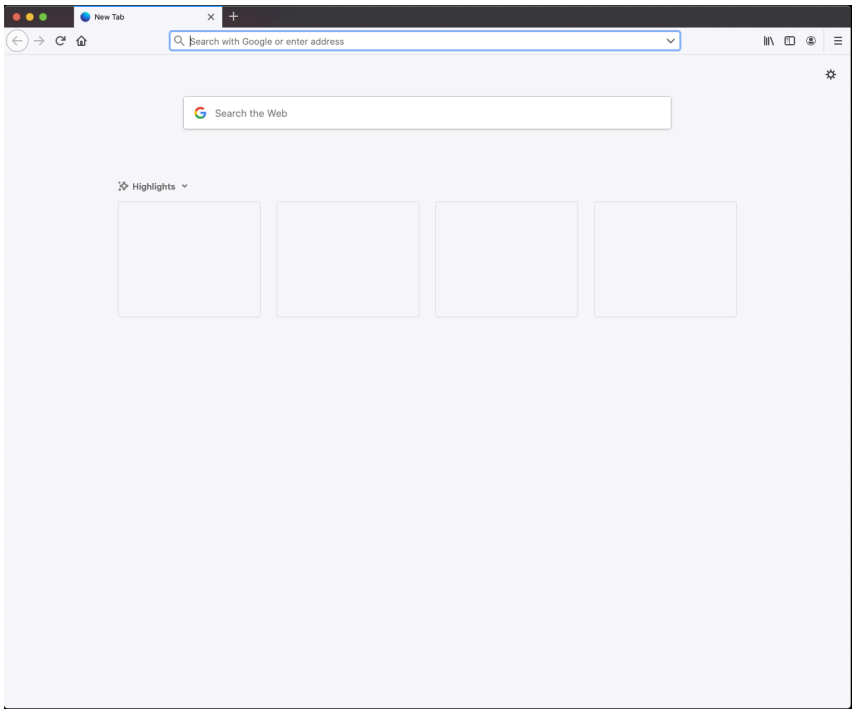


PUBLIC

SECTION_UPDATE

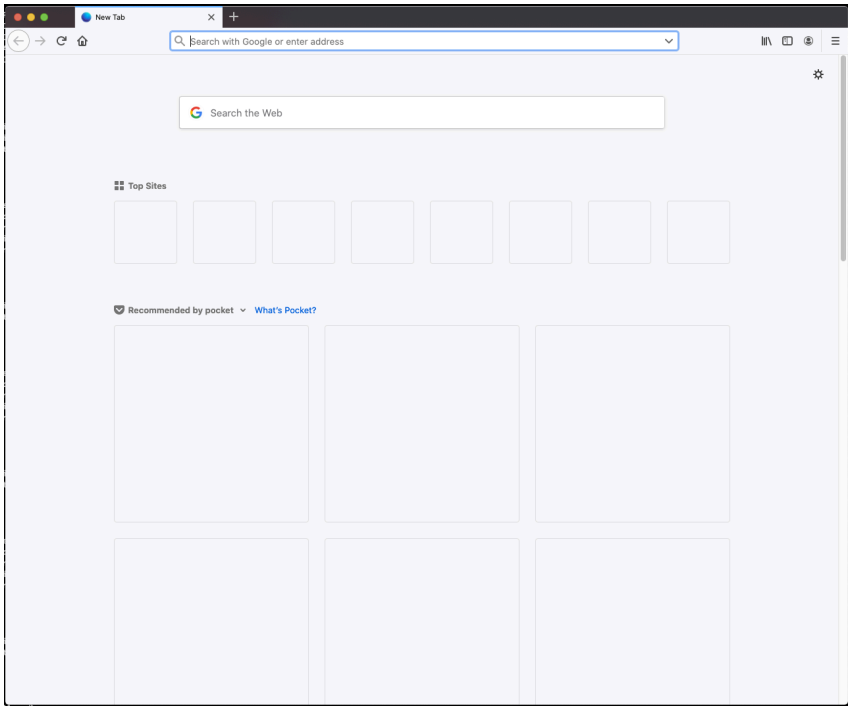


SECTION_UPDATE

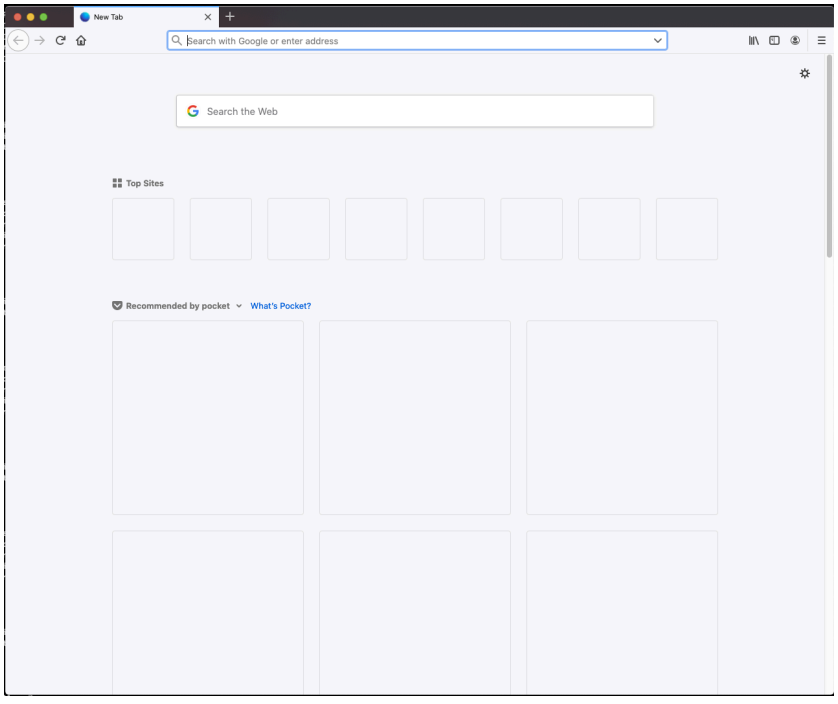


PUBLIC

DISCOVERY_STREAM_LAYOUT_UPDATE

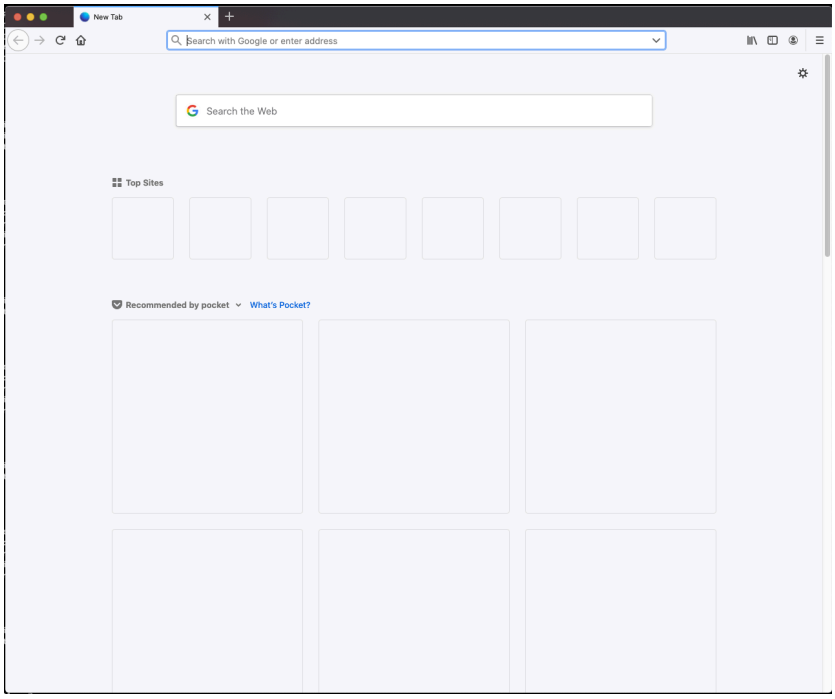


DISCOVERY_STREAM_SPOCS_ENDPOINT

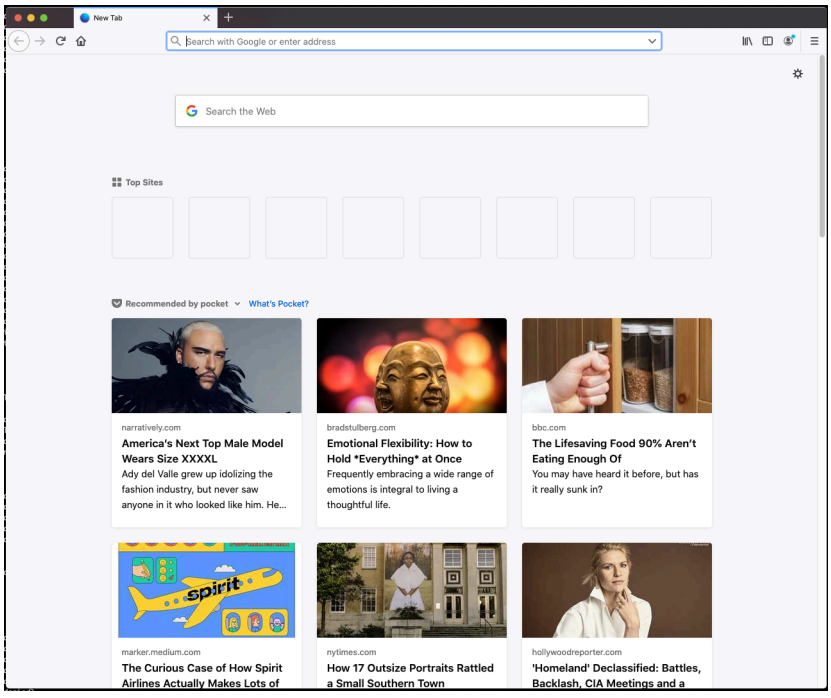


PUBLIC

DISCOVERY_STREAM_SPOCS_UPDATE

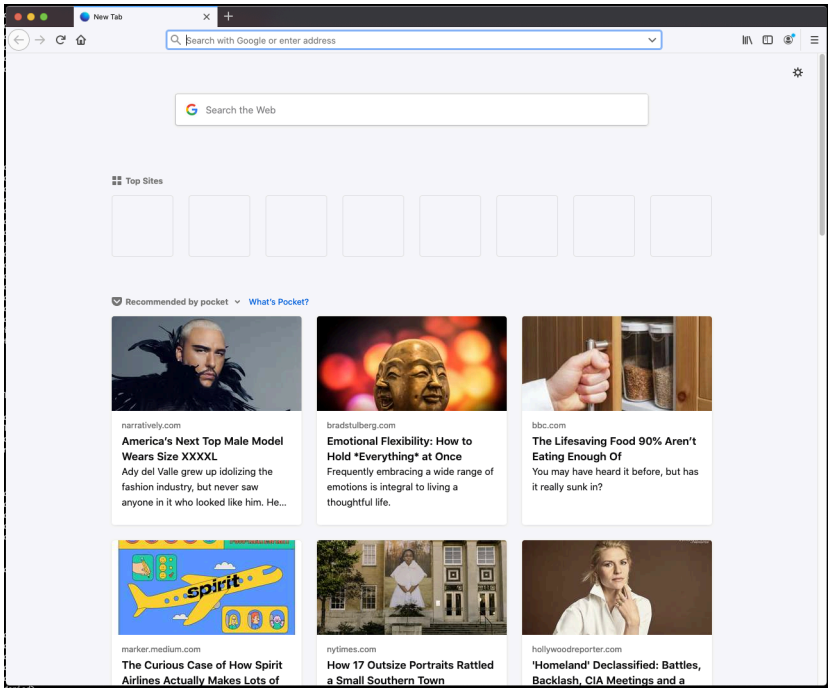


DISCOVERY_STREAM_FEED_UPDATE

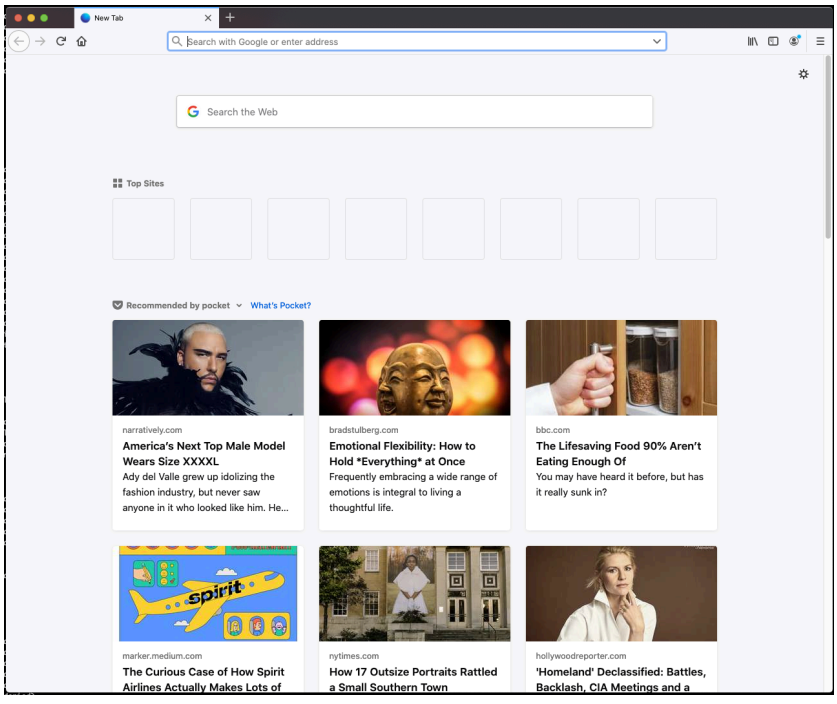


PUBLIC

DISCOVERY_STREAM_FEEDS_UPDATE

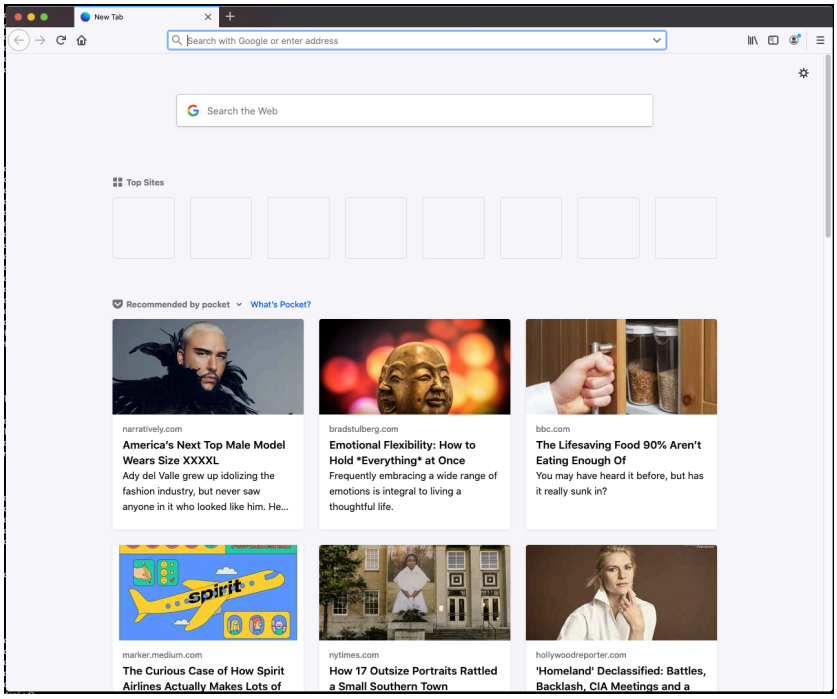


UPDATE_SEARCH_SHORTCUTS

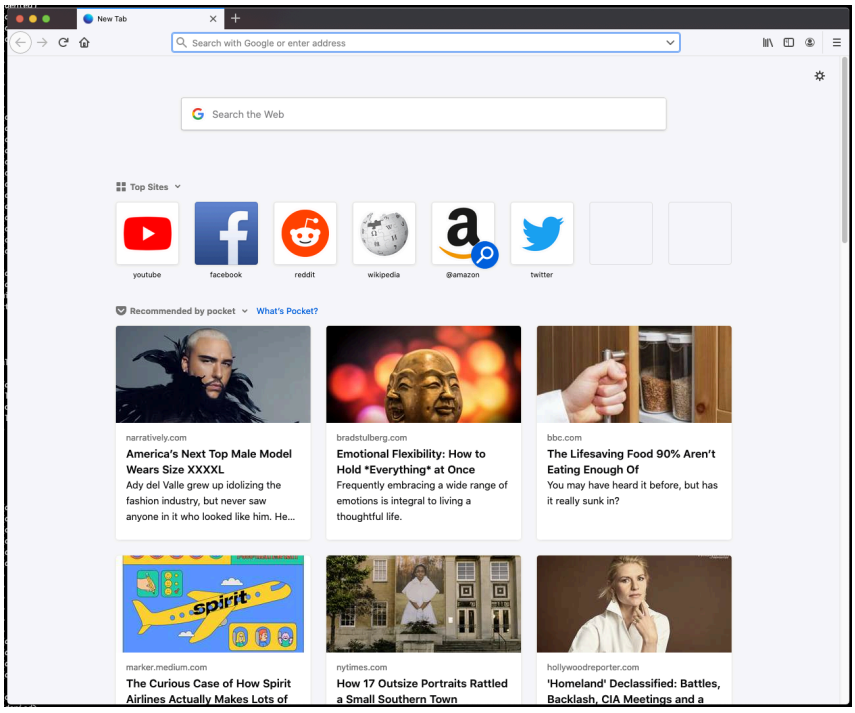


PUBLIC

SAVE_SESSION_PERF_DATA

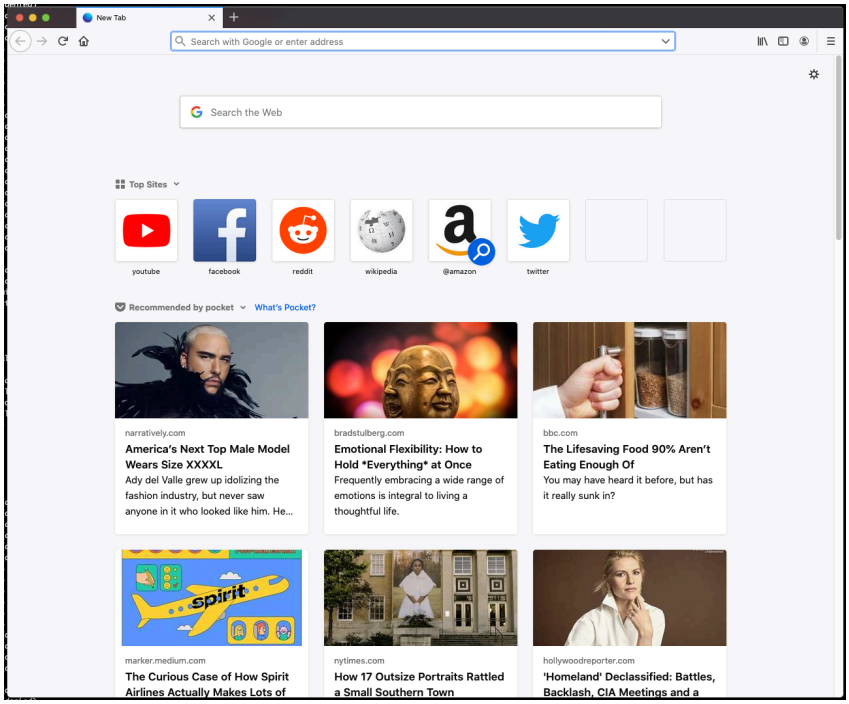


TOP_SITES_UPDATED

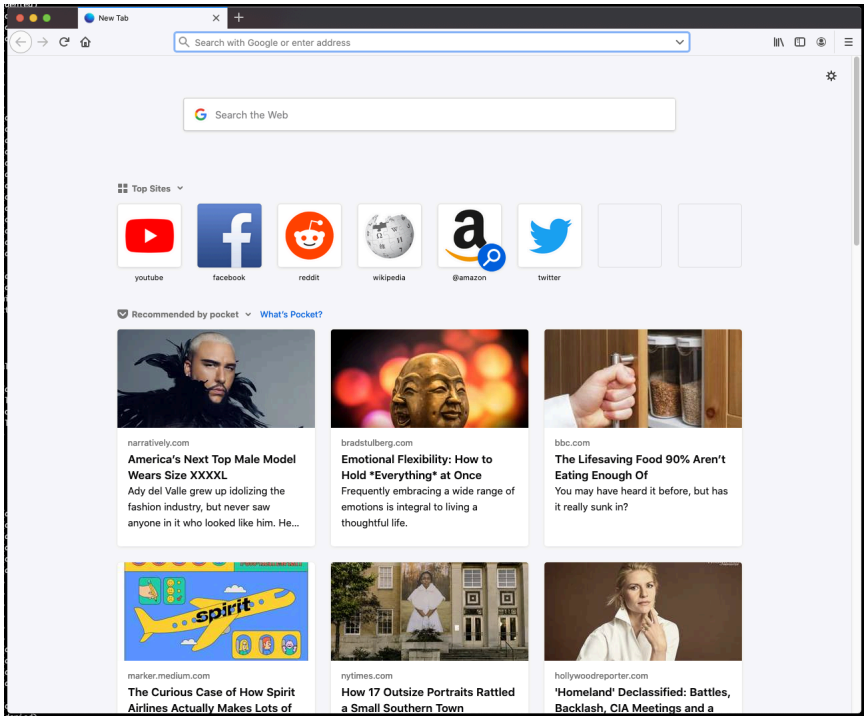


PUBLIC

SECTION_UPDATE



AS_ROUTER_INITIALIZED



Here, we consider the presentation of `about:home` to be complete.

Startup variations

The previous section walks the reader through how about:home starts up in the “pre-existing profile startup, no automated session restore” common case.

These are some variations that might occur based on user configuration, or the point within the browser’s lifecycle that it occurs.

First startup - a brand new profile is being started

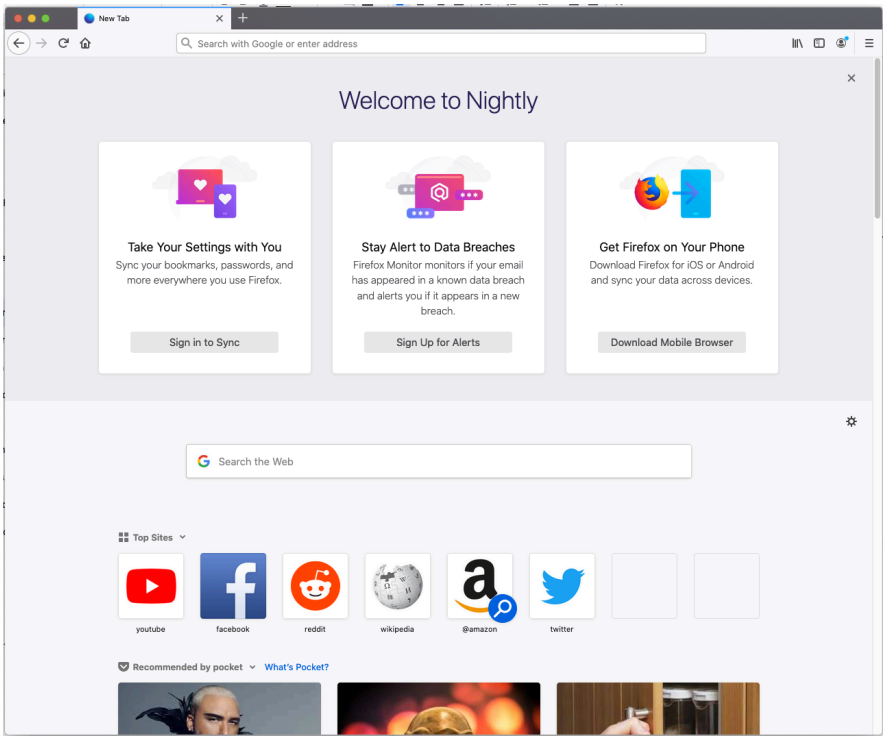
In this variation, all of the caches are empty, so any effort that about:home makes on using cached data to speed up initial presentation fails. This means that we need to make network requests to populate the Pocket stories with text and images.

We do not, however, need to make network requests for the default Top Sites, since the high-resolution favicons are shipped with Firefox.

The first start scenario is complicated by the fact that *all* of Firefox, and not just the code of `about:home`, is grappling with the fact that it’s running for a profile for the very first time. This means very different operating characteristics. Particularly, it’s expected that the disk and CPU will be busier since it won’t have the advantage of precomputed caches in the profile directory.

Focusing on about:home, however, the main visual difference is that a variation of about:home is presented:

PUBLIC



What's particularly interesting is that, Pocket stories and localized strings aside, this initial `about:home` document is virtually the same for every user that starts up a fresh profile.

Post-upgrade startup - the browser is starting, having just completed an upgrade

In this scenario, the startupcache will have been invalidated, so the CPU and disk will probably be a bit busier grappling with the lack of valid cache to work with. This means that about:home should do its best to avoid being blocked by the disk, and to reduce the CPU required to compute the initial about:home document to avoid being slowed down in this scenario.

It does look as if there have been efforts over time to migrate settings, or apply experiments to user profiles after an upgrade, but I'm not aware of any such experiments that are currently underway.

So beyond the startupcache, I'm not sure this scenario differs much from the common startup case.

about:home pages loaded in new windows

This scenario skips the Activity Stream initialization step. Instead, upon loading the about:home document, the state is requested from the parent, and then the parent sends down the actions necessary to render about:home. As noted earlier, the number of actions required is quite high.

Preloaded about:newtab load

As an optimization for opening new tabs, when the main thread is idle for a sufficient period of time, a hidden about:newtab page is loaded in a hidden browser in the background. If the user ever attempts to open a new tab such that about:newtab would be loaded, the backgrounded "preloaded" about:newtab swaps in for the newly opened tab. This results in a significant perceived performance win when opening tabs, as the user never sees the page actually load, and instead it appears fully formed.

This preloaded about:newtab is constructed in the same way as any other about:home document, however it is kept updated by the parent process about various changes in the parent process Redux state. This is to make sure that the preloaded about:newtab is as up-to-date as possible[9].

Otherwise, this scenario doesn't seem significantly different.

Findings

Rendering the initial about:home to any meaningful degree requires considerable disk and CPU time in the parent process during the startup window

If the reader takes anything away from this document, it's that there is a considerable amount of machinery that starts up in order to provide the state object that's ultimately used to render the initial about:home.

Producing that state requires accessing information from a variety of disk and database sources and sometimes network sources, and then applying transformations on that data to produce the final state. The data accesses and the transformations require a non-trivial amount of disk and CPU activity in a window time (application startup) when disk and CPU are precious resources that should be used sparingly.

I'll note that the CPU usage is often spread out over many ticks of the event loop due to a liberal usage of Promises and async functions. This is good for keeping the event loop responsive, but is less good for producing that initial state object in a timely manner, as other events in the event loop can delay its creation or delivery.

[9] To be clear, this is to avoid the problem where a preloaded about:newtab sits idle for so long that by the time it's used, it's information is woefully out of date.

The Redux Store in the parent process waits for its IndexedDB connection to be available before state requests from content are responded to

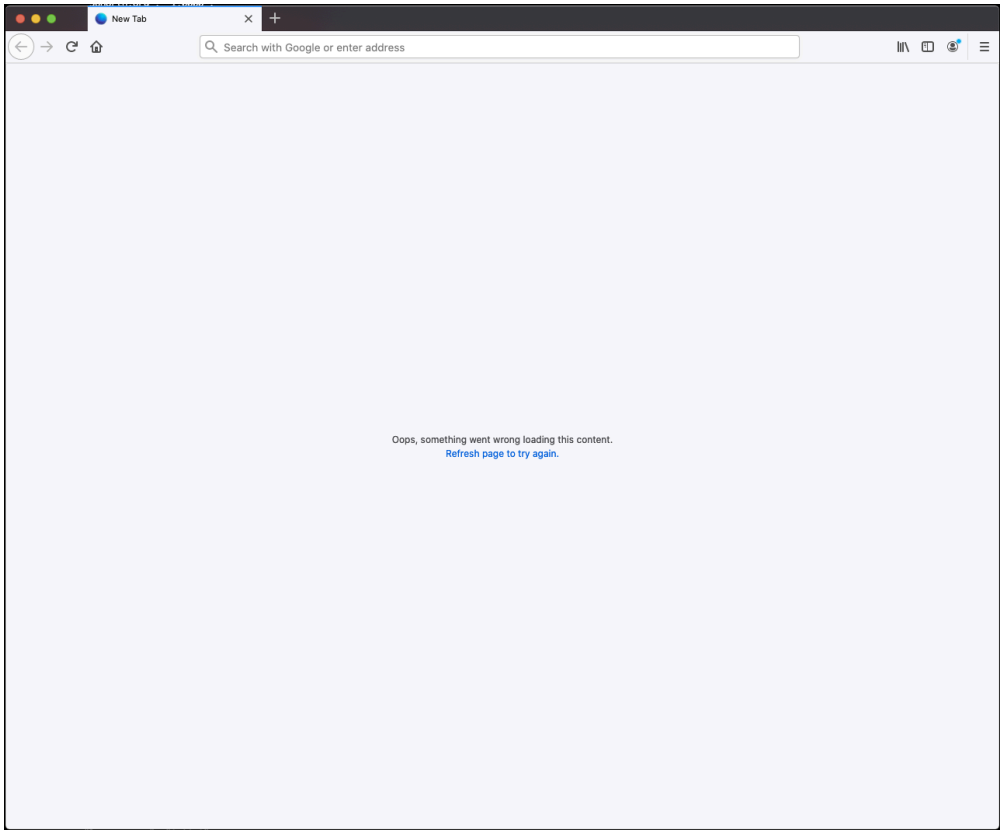
Refer back to the point when the parent process waits for an IndexedDB connection to be opened while initting the parent process Redux store.

This is an interesting point of time from a performance point of view, because we're essentially at a standstill waiting for the IndexedDB database backend to start and to return us a connection object for us to work with. This could be any amount of time.

While we're waiting, the `ActivityStreamMessageChannel` has not yet been initialized, meaning that we're ignoring messages from about:home documents that might finish loading long before the IndexedDB database connection becomes available. Even if we reorganize the `init` function to init the channel sooner, the `NewTabInit` feed will not be initialized, so state requests will be ignored. If we move the `NewTabInit` feed initialization to occur sooner, the default state is sent down.

The default state doesn't result in anything being displayed

Even if we perform the modifications mentioned in the previous paragraph to the Store initialization method, and even if the default state reaches the initial about:home document, nothing is displayed with that default state:



So presumably we need to populate that state with more information before we can render a single pixel and show the user any progress.

It requires several state changes before about:home has enough information to render completely

See this section for more detail on which actions are computed and in what order, as well as what ends up being rendered for each action.

Since we're trying to optimize displaying the initial about:home, this looks like an fruitful area to tighten up - ideally, if we need to compute the document dynamically using a state object, that object should be available as soon as possible in such a way to render meaningful content to the user.

There are frequent writes to disk to persist caching data that could be made lazier

There appears to be frequent writes to disk by way of the `PersistentCache` class, where every time a cache is updated, it is immediately written to disk.

In the worst case, writing the disk seeks the disk head such that subsequent reads from the disk take longer to perform. Disk IO should only be used when necessary.

Recommendations

Given that about:home as a feature has already shipped for at least some of our users, it doesn't seem helpful to mark any recommendations as "Blocking". Therefore, the recommendations below will be given a classification of:

- **Important (5)**
- **High Priority (3)**
- **Suggestion (1)**

Don't rely on dynamically computed state in order to render the initial about:home

Classification

Important

Details

In order to render any significant chunk of the initial about:home, it's necessary for the relevant feeds and parent process Redux store to be instantiated in the parent process, and for the state object to be sent back down to the initial tab so that React can convert it into DOM to be painted.

This is a lot to do during the startup window, and I suspect that we can save a lot of time rendering the first about:home by loading it as a static document rather than by generating it dynamically from the state sent down from the parent.

This means periodically flushing the most recent about:home document and state object to disk to be read during the next startup.

The plan to do this is long and complicated, and so I've split it out to its own document.

Construct the initial new profile about:home document at build-time

Classification

Important

PUBLIC

Details

For brand new profiles, the initial appearance of the first about:home is a known quantity - we know what the default set of top sites are, we know what welcome text we might want to display to the user, etc. Since we know all of these things, it should be possible to construct the initial about:home statically at build-time so that the initial about:blank does not need to be generated dynamically.

The Pocket stories and topics, however, would still need to populate dynamically.

Calculating the majority of the initial about:home statically would free up a considerable amount of CPU and disk resources that can be used for other first-run operations.

Move as much computation of about:home state out of the parent process as possible

Classification

Important

Details

The main thread in the parent process is what services UI events from the user. Any lag in that thread results can result in dropped frames or poor / no responsiveness to user input. In the worst case, the main thread becomes completely stuck and the entire browser application becomes unresponsive.

In general, moving operations out of the parent process main thread is a win for responsiveness [10]. The feeds that mutate the parent process Redux state all execute in the parent process main thread. I believe these can and should be modified to run in a content process main thread.

The plan to do this is long and complicated, and so I've split it out to its own document.

Avoid excessive JSON parsing / serialization, and excessive writing to the preferences database and disk

Classification

Important

[10] It's usually a win, but it's not a guaranteed one. Multiple threads of execution don't represent free computation, and we are at the mercy of the operating system thread scheduler, so the main thread could still be starved for CPU time.

PUBLIC

Details

It appears as if there's a branch of Discovery Stream's prefs that stores a JSON object. This object is often parsed, modified, and then reserialized, all on the main thread.

For example, every `set` of `LinksStorage` results in JSON serialization of an Object to a string, and then a write to the preferences database.

Instead of writing so aggressively, it would be better to synchronously update an `Object` representing the state, and then queuing a low-priority task to eventually write that state to disk, coalescing or debouncing clusters of modifications to the `Object`.

Ultimately, in the ideal case, the `LinksStorage` and `PersistentCache` class would only ever write to the preferences database or disk when the browser is shutting down, or when idle, and would otherwise act as an in-memory cache.

This is somewhat related to the `JSONFile` suggestion further below, though could also be remedied without `JSONFile`.

Consult early and often with the front-end performance team on changes to about:home

Classification

Important

Details

`about:home` is clearly an important piece of UI in the browser, with plenty of stakeholders from across the organization. The earlier that the front-end performance team can be consulted and involved in the design of new features or architecture changes, the easier it is for us to provide guidance that doesn't require major refactorings or stress near ship-dates.

If there's a periodic development standup that the about:home engineers do, it might be worth having one of the front-end performance team members sit in and act as a liaison / consultant.

Avoid messaging overhead by reading from the preferences database directly from a content process

Classification

High Priority

PUBLIC

Details

The React code in the content process uses its copy of the parent process state object to produce the final about:home document. Various preferences are stored in that state object in the parent process. Updates to those preferences can result in messages sent down to the content process so that they can update their state for the new preference values.

This is somewhat redundant - the preferences database can be read directly from the content process, and when the preference is changed, each content process is able to observe those updates using the nsIPrefBranch observation mechanism. The current approach to update each content process about preference changes from PrefsFeed is therefore somewhat redundant and adds additional IPC overhead.

Avoid masking performance impacts by queuing functions to run after the point at which a performance test stops recording

Classification

High Priority

Details

In this pull request, it seems that a sessionrestore Talos regression was addressed by moving the work for initializing ActivityStream just outside of the `sessionrestore-windows-restored` notification. While the intentions are pure, the effect is that the performance impact of instantiating the ActivityStream class is hidden from the sessionrestore test, but the actual user impact of taking up time during startup remains.

We should avoid trying to hide performance impacts like this. If the performance impact is unavoidable, we should just re-baseline.

Switch from PersistentCache to JSONFile

Classification

High Priority

Details

`PersistentCache` is used to persist the DiscoveryStream and TopStories internal caches to a JSON file in the profile directory. JSONFile performs a similar function, but has better support for

ensuring that the file is written to during shutdown. It also benefits from more testing across more components (Form Auto-fill, Password Manager, WebExtension API, DOM Manifest, etc).

JSONFile also uses DeferredTask so that frequent updates are coalesced, which can reduce IO.

Remove unused Action types

Classification

Suggestion

Details

There are a number of action types that don't appear to be dispatched anymore, and are ultimately contributing to dead code. For simplification, these should be removed. See the action catalog for a list of action types that can probably be removed.

Changelog

1. 2019-11-25 - Analysis started
2. 2019-12-02 - Interview with Kate Hudson
3. 2020-01-27 - First draft completed

Appendix

Questions and answers

What function does the Redux store in the parent process serve?

It's the "source of truth" for the entire application state. For example, the PrefsFeed monitors a set of preferences, and takes their initial state and tosses them into the overall application state. When it observes that any of these preferences have updated (for example, via about:config), that change is turned into a Redux Action and fed into the store to update the internal state. And in some cases, those preference changes result in a message being broadcast to content.

What are the things that consume the main process Redux store state to render UI?

- AboutPreferences.jsm (constructing the about:preferences UI for newtab)
- NewTabInit (copies the state and sends it down to each about:home window scope to render the UI)
- ASRouter.jsm uses some of it to populate CFR and What's New Panels

It seems like the rest of the users of the parent process Redux state read it to make decisions based on that state.

What kind of communication occurs between the parent and content processes for about:home over ActivityStreamMessageChannel?

It seems that there are a discrete set of Actions that can be sent through any of the Redux stores, and each one can be decorated to be sent to one or more Redux stores using these Action Creators.

I've started to catalog the actions here.

When is about:home expected to mutate? How is it expected to mutate? How about for the preloaded browsers?

The preloaded browser should always have the most up-to-date state, so it is constantly mutating as the Redux state in the parent changes.

It appears that there's no distinction between foreground and background tabs when updating. It appears that either all about:home tabs are updated, or only one is (there's only one instance of this, which is when the parent is responding to a tab's request for initial state). There's no checking for the visibility state, for example, when determining whether or not to update the page.

There does appear to be a distinction drawn between currently existing about:home tabs, and future about:home tabs. Pocket stories, for example, choose either to update the state of all tabs, or only update the parent process state (so that subsequently opened new tabs get the new state).

Currently existing about:home instances appear to update immediately for things like Places updates (bookmarking, history), Pocketing of one of the Top Stories, modifying any of the Top Sites, and updating any of the Sections on the page via about:preferences.

Currently existing about:home instances appear to *not* be updated for things like:

1. Sponsored content has been displayed too many times
2. TopSites being updated on a system tick
3. Highlights having been updated due to some Pocket'ing.

The visibility state is not used so that about:home knows whether or not to fully update. Instead, the parent chooses which browsers to send actions to.

An updating policy should probably be crafted and enforced here, since it seems a little inconsistent.

What's the difference between Discovery Stream and Activity Stream?

Activity Stream was developed first. Discovery Stream was developed when the Pocket team was given ownership over about:home, and they expressed a need for being able to quickly experiment on page layout. Discovery Stream was developed as a subcomponent ("feed") of Activity Stream, but which ultimately took over the layout management of "Top Sites", "Pocket" and "Highlights" sections of about:home.

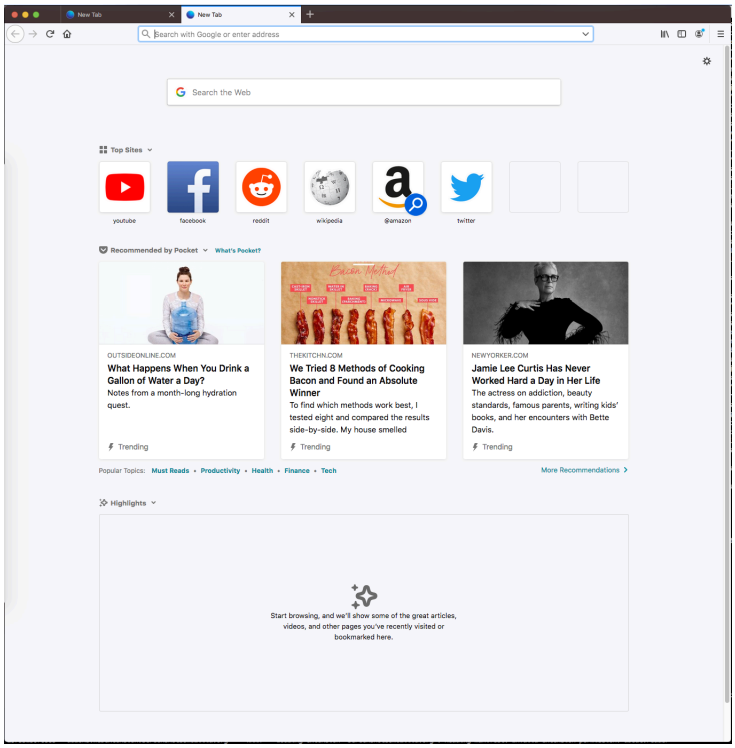
Discovery Stream allows the layout of the page to be controlled remotely, as the DiscoveryStreamFeed can query a WebAPI endpoint to get instructions on how to lay things out. This experimentation appears to have finished though, and the layout is currently hard-coded.

Right now, they co-exist. Some of our users have Activity Stream, and some of them (mainly US and Canadian English builds, and recently de builds) have Discovery Stream enabled by default. It seems likely that eventually we'll have Discovery Stream enabled for everybody, but it's going to be a slow rollout as Pocket makes their stories more relevant for those regions.

This means that any optimization we make should work for both the Activity Stream and Discovery Stream modes of operation.

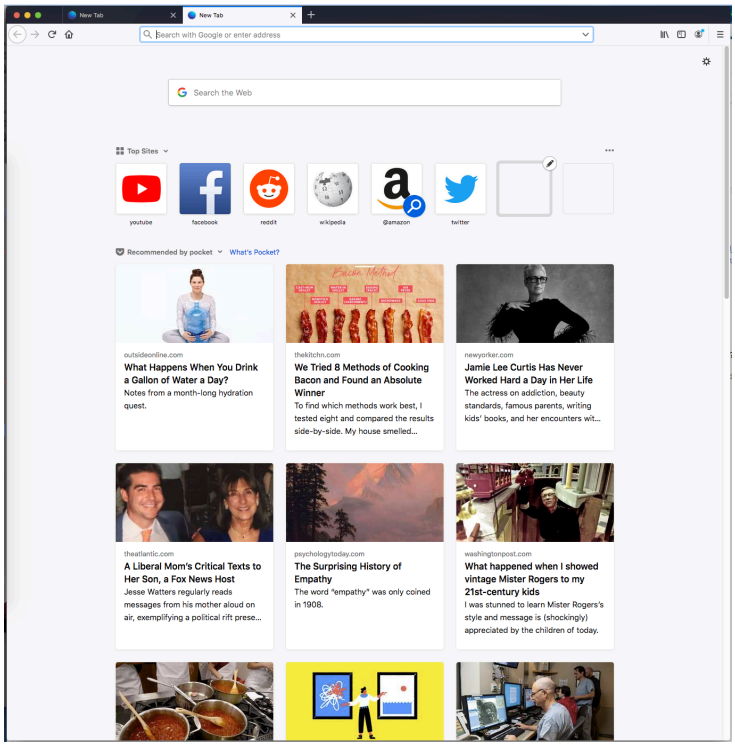
PUBLIC

Appearance



The above is Activity Stream

PUBLIC



This above is Discovery Stream

What is the relationship between about:home and ASRouter? What is their relationship to CFR?

ASRouter was taken on by the team that originally built Activity Stream as a way of communicating to the user about things based on what the current running instance knows about the user. These communications include things like the “What’s New” panel information, as well as contextual communications (regions within the Enhanced Tracking Protection panel, for example). These communications also include “contextual feature recommender” (CFR), like the Recommendation panel that shows up in the AwesomeBar periodically to recommend a feature or an add-on.

ASRouter is mostly distinct from about:home. There are some architectural similarities, and ASRouter uses Activity Stream to initialize itself. It also uses Activity Stream’s TelemetryFeed for feeding information into “Ping Center”.

It’s likely that, over time, ASRouter will grow more separate from the components running about:home, and no longer have any of those dependencies.

So while we might want to consider the impacts any of our recommendations have on ASRouter / CFR, we should be aware that they're very distinct features, and that ASRouter and CFR improvements are out of scope for this analysis, except where they overlap with about:home.

What kinds of storage mechanisms does about:home use, and for what?

IndexedDB

An IndexedDB database is used by `ActivityStreamStorage`, which is used to store things like:

- Section collapsed state
- Other section preferences, for custom sections from WebExtensions (unused)
- ASRouter storage (session information, impressions, blocked messages and providers)

Preferences

PersistentCache

DiscoveryStreamFeed uses this for storing caches of:

- Domain affinites
- The current layout (unused currently, since the layout is hardcoded)
- Feeds information (I believe this is mainly Pocket recommendations)
- Sponsored content

Places

What's the minimum amount of state information that about:home needs in order to render something meaningful?

What information sources ultimately feed into the state used to render about:home?

What does each "Feed" do, and can they easily be moved into a content process?

Chatting with Kate Hudson

- *Is my conception of the messaging model correct?*
    - Yes
- *Is my conception of Activity Stream vs Discovery Stream correct?*
    - Remotely configured layout
    - Unclear whether or not the configurable layouts are still a requirement now
    - *What is a DiscoveryStream "feed"?*

        - Layout feed, content feeds
        - Layout feed references other feeds
        - Layout is now hardcoded
- *Is about:home expected to change after it has been rendered and displayed to the user?*
    - When a user opens about:home, it _should_ be the most up to date
    - There is logic to avoid problems where the UI updates interrupts current activity
    - A “block” action should happen immediately, for example. Any UI action initted from a tab should happen immediately, and for all tabs.
    - If there are 4 windows open, any action initted within one window
    - *What about the preloaded about:home?*
        - It needs to be kept up to date.
- *What’s the relationship between about:home and ASRouter? And CFR?*
    - ASRouter has no Redux, only kinda related, uses ActivityStream for initting, and for feeding Telemetry in, but is going to likely separate further.
- *What’s coming up that I need to be aware of?*
    - Major change is to move ASRouter and CFR stuff out from its dependencies on ActivityStream.
    - Switching to Normandy for experimentation