doctut

Release 0.0.1

keiko

Apr 15, 2021

CONTENTS

This document describes the implementation of Firefox's address bar, also known as the quantumbar or urlbar. The address bar was also called the awesomebar until Firefox 68, when it was substantially rewritten.

The address bar is a specialized search access point that aggregates data from several different sources, including:

- Places (Firefox's history and bookmarks system)
- Search engines (including search suggestions)
- WebExtensions
- Open tabs

Most of the address bar code lives in browser/components/urlbar. A separate and important back-end piece currently is toolkit/components/places/UnifiedComplete.jsm, which was carried over from awesomebar and has not yet been rewritten for quantumbar.

CHAPTER

ONE

ARCHITECTURE OVERVIEW

The address bar is implemented as a *model-view-controller* (MVC) system. One of the scopes of this architecture is to allow easy replacement of its components, for easier experimentation.

Each search is represented by a unique object, the *UrlbarQueryContext*. This object, created by the *View*, describes the search and is passed through all of the components, along the way it gets augmented with additional information. The *UrlbarQueryContext* is passed to the *Controller*, and finally to the *Model*. The model appends results to a property of *UrlbarQueryContext* in chunks, it sorts them through a *Muxer* and then notifies the *Controller*.

See the specific components below, for additional details about each one's tasks and responsibilities.

1.1 The UrlbarQueryContext

The *UrlbarQueryContext* object describes a single instance of a search. It is augmented as it progresses through the system, with various information:

```
UrlbarQueryContext {
  allowAutofill; // {boolean} If true, providers are allowed to return
                 // autofill results.  Even if true, it's up to providers
                 // whether to include autofill results, but when false, no
                 // provider should include them.
  isPrivate; // {boolean} Whether the search started in a private context.
  maxResults; // {integer} The maximum number of results requested. It is
              // possible to request more results than the shown ones, and
              // do additional filtering at the View level.
  searchString; // {string} The user typed string.
  userContextId; // {integer} The user context ID (containers feature).

  // Optional properties.
  muxer; // {string} Name of a registered muxer. Muxers can be registered
         // through the UrlbarProvidersManager.
  providers; // {array} List of registered provider names. Providers can be
             // registered through the UrlbarProvidersManager.
  sources: {array} list of accepted UrlbarUtils.RESULT_SOURCE for the context.
          // This allows to switch between different search modes. If not
          // provided, a default will be generated by the Model, depending on
          // the search string.
  engineName: // {string} if sources is restricting to just SEARCH, this
              // property can be used to pick a specific search engine, by
              // setting it to the name under which the engine is registered
              // with the search service.
  currentPage: // {string} url of the page that was loaded when the search
               // began.
```

(continues on next page)

(continued from previous page)

```
  allowSearchSuggestions: // {boolean} Whether to allow search suggestions.
                          // This is a veto, meaning that when false,
                          // suggestions will not be fetched, but when true,
                          // some other condition may still prohibit
                          // suggestions, like private browsing mode. Defaults
                          // to true.

  // Properties added by the Model.
  results; // {array} list of UrlbarResult objects.
  tokens; // {array} tokens extracted from the searchString, each token is an
          // object in the form {type, value, lowerCaseValue}.
}
```

1.2 The Model

The *Model* is the component responsible for retrieving search results based on the user's input, and sorting them accordingly to their importance. At the core is the UrlbarProvidersManager, a component tracking all the available search providers, and managing searches across them.

The *UrlbarProvidersManager* is a singleton, it registers internal providers on startup and can register/unregister providers on the fly. It can manage multiple concurrent queries, and tracks them internally as separate *Query* objects.

The *Controller* starts and stops queries through the *UrlbarProvidersManager*. It's possible to wait for the promise returned by *startQuery* to know when no more results will be returned, it is not mandatory though. Queries can be canceled.

Note: Canceling a query will issue an interrupt() on the database connection, terminating any running and future SQL query, unless a query is running inside a *runInCriticalSection* task.

The *searchString* gets tokenized by the UrlbarTokenizer component into tokens, some of these tokens have a special meaning and can be used by the user to restrict the search to specific result type (See the *UrlbarTokenizer::TYPE* enum).

Caution: The tokenizer uses heuristics to determine each token's type, as such the consumer may want to check the value before applying filters.

```
UrlbarProvidersManager {
  registerProvider(providerObj);
  unregisterProvider(providerObj);
  registerMuxer(muxerObj);
  unregisterMuxer(muxerObjOrName);
  async startQuery(queryContext);
  cancelQuery(queryContext);
  // Can be used by providers to run uninterruptible queries.
  runInCriticalSection(taskFn);
}
```

1.2.1 UrlbarProvider

A provider is specialized into searching and returning results from different information sources. Internal providers are usually implemented in separate *jsm* modules with a *UrlbarProvider* name prefix. External providers can be registered as *Objects* through the *UrlbarProvidersManager*. Each provider is independent and must satisfy a base API, while internal implementation details may vary deeply among different providers.

Important: Providers are singleton, and must track concurrent searches internally, for example mapping them by UrlbarQueryContext.

Note: Internal providers can access the Places database through the *PlacesUtils.promiseLargeCacheDBConnection* utility.

```
class UrlbarProvider {
  /**
   * Unique name for the provider, used by the context to filter on providers.
   * Not using a unique name will cause the newest registration to win.
   * @abstract
   */
  get name() {
    return "UrlbarProviderBase";
  }
  /**
   * The type of the provider, must be one of UrlbarUtils.PROVIDER_TYPE.
   * @abstract
   */
  get type() {
    throw new Error("Trying to access the base class, must be overridden");
  }
  /**
   * Whether this provider should be invoked for the given context.
   * If this method returns false, the providers manager won't start a query
   * with this provider, to save on resources.
   * @param {UrlbarQueryContext} queryContext The query context object
   * @returns {boolean} Whether this provider should be invoked for the search.
   * @abstract
   */
  isActive(queryContext) {
    throw new Error("Trying to access the base class, must be overridden");
  }
  /**
   * Gets the provider's priority.  Priorities are numeric values starting at
   * zero and increasing in value.  Smaller values are lower priorities, and
   * larger values are higher priorities.  For a given query, `startQuery` is
   * called on only the active and highest-priority providers.
   * @param {UrlbarQueryContext} queryContext The query context object
   * @returns {number} The provider's priority for the given query.
   * @abstract
   */
  getPriority(queryContext) {
    // By default, all providers share the lowest priority.
    return 0;
  }
  /**
```

(continues on next page)

(continued from previous page)

```
   * Starts querying.
   * @param {UrlbarQueryContext} queryContext The query context object
   * @param {function} addCallback Callback invoked by the provider to add a new
   *        result. A UrlbarResult should be passed to it.
   * @note Extended classes should return a Promise resolved when the provider
   *       is done searching AND returning results.
   * @abstract
   */
  startQuery(queryContext, addCallback) {
    throw new Error("Trying to access the base class, must be overridden");
  }
  /**
   * Cancels a running query,
   * @param {UrlbarQueryContext} queryContext The query context object to cancel
   *        query for.
   * @abstract
   */
  cancelQuery(queryContext) {
    throw new Error("Trying to access the base class, must be overridden");
  }
}
```

1.2.2 UrlbarMuxer

The *Muxer* is responsible for sorting results based on their importance and additional rules that depend on the UrlbarQueryContext. The muxer to use is indicated by the UrlbarQueryContext.muxer property.

> **Caution:** The Muxer is a replaceable component, as such what is described here is a reference for the default View, but may not be valid for other implementations.

```
class UrlbarMuxer {
  /**
   * Unique name for the muxer, used by the context to sort results.
   * Not using a unique name will cause the newest registration to win.
   * @abstract
   */
  get name() {
    return "UrlbarMuxerBase";
  }
  /**
   * Sorts UrlbarQueryContext results in-place.
   * @param {UrlbarQueryContext} queryContext the context to sort results for.
   * @abstract
   */
  sort(queryContext) {
    throw new Error("Trying to access the base class, must be overridden");
  }
}
```

1.3 The Controller

UrlbarController is the component responsible for reacting to user's input, by communicating proper course of action to the Model (e.g. starting/stopping a query) and the View (e.g. showing/hiding a panel). It is also responsible for reporting Telemetry.

Note: Each *View* has a different *Controller* instance.

```
UrlbarController {
  async startQuery(queryContext);
  cancelQuery(queryContext);
  // Invoked by the ProvidersManager when results are available.
  receiveResults(queryContext);
  // Used by the View to listen for results.
  addQueryListener(listener);
  removeQueryListener(listener);
}
```

1.4 The View

The View is the component responsible for presenting search results to the user and handling their input.

1.4.1 UrlbarInput.jsm

Implements an input box *View*, owns an *UrlbarView*.

```
UrlbarInput {
  constructor(options = { textbox, panel });
  // Uses UrlbarValueFormatter to highlight the base host, search aliases
  // and to keep the host visible on overflow.
  formatValue(val);
  openResults();
  // Converts an internal URI (e.g. a URI with a username or password) into
  // one which we can expose to the user.
  makeURIReadable(uri);
  // Handles an event which would cause a url or text to be opened.
  handleCommand();
  // Called by the view when a result is selected.
  resultsSelected();
  // The underlying textbox
  textbox;
  // The results panel.
  panel;
  // The containing window.
  window;
  // The containing document.
  document;
  // An UrlbarController instance.
  controller;
  // An UrlbarView instance.
  view;
```

(continues on next page)

(continued from previous page)

```
  // Whether the current value was typed by the user.
  valueIsTyped;
  // Whether the context is in Private Browsing mode.
  isPrivate;
  // Whether the input box is focused.
  focused;
  // The go button element.
  goButton;
  // The current value, can also be set.
  value;
}
```

1.4.2 UrlbarView.jsm

Represents the base *View* implementation, communicates with the *Controller*.

```
UrlbarView {
  // Manage View visibility.
  open();
  close();
  // Invoked when the query starts.
  onQueryStarted(queryContext);
  // Invoked when new results are available.
  onQueryResults(queryContext);
  // Invoked when the query has been canceled.
  onQueryCancelled(queryContext);
  // Invoked when the query is done. This is invoked in any case, even if the
  // query was canceled earlier.
  onQueryFinished(queryContext);
  // Invoked when the view opens.
  onViewOpen();
  // Invoked when the view closes.
  onViewClose();
}
```

1.5 UrlbarResult

An UrlbarResult instance represents a single search result with a result type, that identifies specific kind of results. Each kind has its own properties, that the *View* may support, and a few common properties, supported by all of the results.

Note: Result types are also enumerated by *UrlbarUtils.RESULT_TYPE*.

```
UrlbarResult {
  constructor(resultType, payload);

  type: {integer} One of UrlbarUtils.RESULT_TYPE.
  source: {integer} One of UrlbarUtils.RESULT_SOURCE.
  title: {string} A title that may be used as a label for this result.
  icon: {string} Url of an icon for this result.
```

(continues on next page)

(continued from previous page)

```
  payload: {object} Object containing properties for the specific RESULT_TYPE.
  autofill: {object} An object describing the text that should be
            autofilled in the input when the result is selected, if any.
  autofill.value: {string} The autofill value.
  autofill.selectionStart: {integer} The first index in the autofill
                           selection.
  autofill.selectionEnd: {integer} The last index in the autofill selection.
  suggestedIndex: {integer} Suggest a preferred position for this result
                  within the result set. Undefined if none.
}
```

The following RESULT_TYPEs are supported:

```
// An open tab.
// Payload: { icon, url, userContextId }
TAB_SWITCH: 1,
// A search suggestion or engine.
// Payload: { icon, suggestion, keyword, query, providesSearchMode, inPrivateWindow,␣
↪isPrivateEngine }
SEARCH: 2,
// A common url/title tuple, may be a bookmark with tags.
// Payload: { icon, url, title, tags }
URL: 3,
// A bookmark keyword.
// Payload: { icon, url, keyword, postData }
KEYWORD: 4,
// A WebExtension Omnibox result.
// Payload: { icon, keyword, title, content }
OMNIBOX: 5,
// A tab from another synced device.
// Payload: { icon, url, device, title }
REMOTE_TAB: 6,
// An actionable message to help the user with their query.
// textData and buttonTextData are objects containing an l10n id and args.
// If a tip is untranslated it's possible to provide text and buttonText.
// Payload: { icon, textData, buttonTextData, [buttonUrl], [helpUrl] }
TIP: 7,
// A type of result created at runtime, for example by an extension.
// Payload: { dynamicType }
DYNAMIC: 8,
```

CHAPTER

TWO

UTILITIES

Various modules provide shared utilities to the other components:

2.1 UrlbarPrefs.jsm

Implements a Map-like storage or urlbar related preferences. The values are kept up-to-date.

```
// Always use browser.urlbar. relative branch, except for the preferences in
// PREF_OTHER_DEFAULTS.
UrlbarPrefs.get("delay"); // Gets value of browser.urlbar.delay.
```

Note: Newly added preferences should always be properly documented in UrlbarPrefs.

2.2 UrlbarUtils.jsm

Includes shared utils and constants shared across all the components.

CHAPTER

THREE

TELEMETRY

This section describes existing telemetry probes measuring interaction with the Address Bar.

3.1 Histograms

PLACES_AUTOCOMPLETE_1ST_RESULT_TIME_MS This probe tracks the amount of time it takes to get the first result. It is an exponential histogram with values between 5 and 100.

PLACES_AUTOCOMPLETE_6_FIRST_RESULTS_TIME_MS This probe tracks the amount of time it takes to get the first six results. It is an exponential histogram with values between 50 and 1000.

FX_URLBAR_SELECTED_RESULT_METHOD This probe tracks how a result was picked by the user from the list. It is a categorical histogram with these values:

- `enter` The user pressed Enter without selecting a result first. This most likely happens when the user confirms the default preselected result (aka *heuristic result*), or when they select with the keyboard a one-off search button and confirm with Enter.
- `enterSelection` The user selected a result, but not using Tab or the arrow keys, and then pressed Enter. This is a rare and generally unexpected event, there may be exotic ways to select a result we didn't consider, that are tracked here. Look at arrowEnterSelection and tabEnterSelection for more common actions.
- `click` The user clicked on a result.
- `arrowEnterSelection` The user selected a result using the arrow keys, and then pressed Enter.
- `tabEnterSelection` The first key the user pressed to select a result was the Tab key, and then they pressed Enter. Note that this means the user could have used the arrow keys after first pressing the Tab key.
- `rightClickEnter` Before QuantumBar, it was possible to right-click a result to highlight but not pick it. Then the user could press Enter. This is no more possible.

3.2 Scalars

urlbar.tips This is a keyed scalar whose values are uints and are incremented each time a tip result is shown, a tip is picked, and a tip's help button is picked. The keys are:

- `intervention_clear-help` Incremented when the user picks the help button in the clear-history search intervention.
- `intervention_clear-picked` Incremented when the user picks the clear-history search intervention.

- intervention_clear-shown Incremented when the clear-history search intervention is shown.
- intervention_refresh-help Incremented when the user picks the help button in the refresh-Firefox search intervention.
- intervention_refresh-picked Incremented when the user picks the refresh-Firefox search intervention.
- intervention_refresh-shown Incremented when the refresh-Firefox search intervention is shown.
- intervention_update_ask-help Incremented when the user picks the help button in the update_ask search intervention, which is shown when there's a Firefox update available but the user's preference says we should ask them to download and apply it.
- intervention_update_ask-picked Incremented when the user picks the update_ask search intervention.
- intervention_update_ask-shown Incremented when the update_ask search intervention is shown.
- intervention_update_refresh-help Incremented when the user picks the help button in the update_refresh search intervention, which is shown when the user's browser is up to date but they triggered the update intervention. We show this special refresh intervention instead.
- intervention_update_refresh-picked Incremented when the user picks the update_refresh search intervention.
- intervention_update_refresh-shown Incremented when the update_refresh search intervention is shown.
- intervention_update_restart-help Incremented when the user picks the help button in the update_restart search intervention, which is shown when there's an update and it's been downloaded and applied. The user needs to restart to finish.
- intervention_update_restart-picked Incremented when the user picks the update_restart search intervention.
- intervention_update_restart-shown Incremented when the update_restart search intervention is shown.
- intervention_update_web-help Incremented when the user picks the help button in the update_web search intervention, which is shown when we can't update the browser or possibly even check for updates for some reason, so the user should download the latest version from the web.
- intervention_update_web-picked Incremented when the user picks the update_web search intervention.
- intervention_update_web-shown Incremented when the update_web search intervention is shown.
- tabtosearch-shown Increment when a non-onboarding tab-to-search result is shown, once per engine per engagement. Please note that the number of times non-onboarding tab-to-search results are picked is the sum of all keys in urlbar.searchmode.tabtosearch. Please also note that more detailed telemetry is recorded about both onboarding and non-onboarding tab-to-search results in urlbar.tabtosearch.*. These probes in urlbar.tips are still recorded because urlbar.tabtosearch.* is not currently recorded in Release.
- tabtosearch_onboard-shown Incremented when a tab-to-search onboarding result is shown, once per engine per engagement. Please note that the number of times tab-to-search onboarding results are picked is the sum of all keys in urlbar.searchmode.tabtosearch_onboard.
- searchTip_onboard-picked Incremented when the user picks the onboarding search tip.

- `searchTip_onboard-shown` Incremented when the onboarding search tip is shown.
- `searchTip_redirect-picked` Incremented when the user picks the redirect search tip.
- `searchTip_redirect-shown` Incremented when the redirect search tip is shown.

urlbar.searchmode.* This is a set of keyed scalars whose values are uints incremented each time search mode is entered in the Urlbar. The suffix on the scalar name describes how search mode was entered. Possibilities include:

- `bookmarkmenu` Used when the user selects the Search Bookmarks menu item in the Library menu.
- `handoff` Used when the user uses the search box on the new tab page and is handed off to the address bar. NOTE: This entry point was deprecated in Firefox 88. Handoff no longer enters search mode.
- `keywordoffer` Used when the user selects a keyword offer result.
- `oneoff` Used when the user selects a one-off engine in the Urlbar.
- `shortcut` Used when the user enters search mode with a keyboard shortcut or menu bar item (e.g. `Accel+K`).
- `tabmenu` Used when the user selects the Search Tabs menu item in the tab overflow menu.
- `tabtosearch` Used when the user selects a tab-to-search result. These results suggest a search engine when the search engine's domain is autofilled.
- `tabtosearch_onboard` Used when the user selects a tab-to-search onboarding result. These are shown the first few times the user encounters a tab-to-search result.
- `topsites_newtab` Used when the user selects a search shortcut Top Site from the New Tab Page.
- `topsites_urlbar` Used when the user selects a search shortcut Top Site from the Urlbar.
- `touchbar` Used when the user taps a search shortct on the Touch Bar, available on some Macs.
- `typed` Used when the user types an engine alias in the Urlbar.
- `other` Used as a catchall for other behaviour. We don't expect this scalar to hold any values. If it does, we need to correct an issue with search mode entry points.

The keys for the scalars above are engine and source names. If the user enters a remote search mode with a built-in engine, we record the engine name. If the user enters a remote search mode with an engine they installed (e.g. via OpenSearch or a WebExtension), we record `other` (not to be confused with the `urlbar.searchmode.other` scalar above). If they enter a local search mode, we record the English name of the result source (e.g. "bookmarks", "history", "tabs"). Note that we slightly modify the engine name for some built-in engines: we flatten all localized Amazon sites (Amazon.com, Amazon.ca, Amazon.de, etc.) to "Amazon" and we flatten all localized Wikipedia sites (Wikipedia (en), Wikipedia (fr), etc.) to "Wikipedia". This is done to reduce the number of keys used by these scalars.

urlbar.picked.* This is a set of keyed scalars whose values are uints incremented each time a result is picked from the Urlbar. The suffix on the scalar name is the result type. The keys for the scalars above are the 0-based index of the result in the urlbar panel when it was picked.

Note: Available from Firefox 84 on. Use the *FX_URLBAR_SELECTED_** histograms in earlier versions. See the *Obsolete probes* section below.

Valid result types are:

- `autofill` An origin or a URL completed the user typed text inline.
- `bookmark` A bookmarked URL.

- `dynamic` A specially crafted result, often used in experiments when basic types are not flexible enough for a rich layout.
- `extension` Added by an add-on through the omnibox WebExtension API.
- `formhistory` A search suggestion from previous search history.
- `history` A URL from history.
- `keyword` A bookmark keyword.
- `remotetab` A tab synced from another device.
- `searchengine` A search result, but not a suggestion. May be the default search action or a search alias.
- `searchsuggestion` A remote search suggestion.
- `switchtab` An open tab.
- `tabtosearch` A tab to search result.
- `tip` A tip result.
- `topsite` An entry from top sites.
- `unknown` An unknown result type, a bug should be filed to figure out what it is.
- `visiturl` The user typed string can be directly visited.

urlbar.picked.searchmode.* This is a set of keyed scalars whose values are uints incremented each time a result is picked from the Urlbar while the Urlbar is in search mode. The suffix on the scalar name is the search mode entry point. The keys for the scalars are the 0-based index of the result in the urlbar panel when it was picked.

Note: These scalars share elements of both `urlbar.picked.*` and `urlbar.searchmode.*`. Scalar name suffixes are search mode entry points, like `urlbar.searchmode.*`. The keys for these scalars are result indices, like `urlbar.picked.*`.

Note: These data are a subset of the data recorded by `urlbar.picked.*`. For example, if the user enters search mode by clicking a one-off then selects a Google search suggestion at index 2, we would record in **both** `urlbar.picked.searchsuggestion` and `urlbar.picked.searchmode.oneoff`.

---

urlbar.tabtosearch.* This is a set of keyed scalars whose values are uints incremented when a tab-to-search result is shown, once per engine per engagement. There are two sub-probes: `urlbar.tabtosearch.impressions` and `urlbar.tabtosearch.impressions_onboarding`. The former records impressions of regular tab-to-search results and the latter records impressions of onboarding tab-to-search results. The key values are identical to those of the `urlbar.searchmode.*` probes: they are the names of the engines shown in the tab-to-search results. Engines that are not built in are grouped under the key `other`.

Note: Due to the potentially sensitive nature of these data, they are currently collected only on pre-release version of Firefox. See bug 1686330.

3.3 Event Telemetry

The event telemetry is grouped under the `urlbar` category.

Event Method There are two methods to describe the interaction with the urlbar:

- `engagement` It is defined as a completed action in urlbar, where a user inserts text and executes one of the actions described in the Event Object.
- `abandonment` It is defined as an action where the user inserts text but does not complete an engagement action, usually unfocusing the urlbar. This also happens when the user switches to another window, regardless of urlbar focus.

Event Value This is how the user interaction started

- `typed`: The text was typed into the urlbar.
- `dropped`: The text was drag and dropped into the urlbar.
- `pasted`: The text was pasted into the urlbar.
- `topsites`: The user opened the urlbar view without typing, dropping, or pasting. In these cases, if the urlbar input is showing the URL of the loaded page and the user has not modified the input's content, the urlbar views shows the user's top sites. Otherwise, if the user had modified the input's content, the urlbar view shows results based on what the user has typed. To tell whether top sites were shown, it's enough to check whether value is `topsites`. To know whether the user actually picked a top site, check check that `numChars` == 0. If `numChars` > 0, the user initially opened top sites, but then they started typing and confirmed a different result.
- `returned`: The user abandoned a search, for example by switching to another tab/window, or focusing something else, then came back to it and continued. We consider a search continued if the user kept at least the first char of the original search string.
- `restarted`: The user abandoned a search, for example by switching to another tab/window, or focusing something else, then came back to it, cleared it and then typed a new string.

Event Object These describe actions in the urlbar:

- `click` The user clicked on a result.
- `enter` The user confirmed a result with Enter.
- `drop_go` The user dropped text on the input field.
- `paste_go` The user used Paste and Go feature. It is not the same as paste and Enter.
- `blur` The user unfocused the urlbar. This is only valid for `abandonment`.

Event Extra This object contains additional information about the interaction. Extra is a key-value store, where all the keys and values are strings.

- `elapsed` Time in milliseconds from the initial interaction to an action.
- `numChars` Number of input characters the user typed or pasted at the time of submission.
- `numWords` Number of words in the input. The measurement is taken from a trimmed input split up by its spaces. This is not a perfect measurement, since it will return an incorrect value for languages that do not use spaces or URLs containing spaces in its query parameters, for example.
- `selType` The type of the selected result at the time of submission. This is only present for `engagement` events. It can be one of: `none`, `autofill`, `visiturl`, `bookmark`, `history`, `keyword`, `searchengine`, `searchsuggestion`, `switchtab`, `remotetab`, `extension`, `oneoff`, `keywordoffer`, `canonized`, `tip`, `tiphelp`, `formhistory`, `tabtosearch`, `help`, `unknown` In practice, `tabtosearch` should not appear in real event telemetry. Opening a tab-to-search

result enters search mode and entering search mode does not currently mark the end of an engagement. It is noted here for completeness.

- `selIndex` Index of the selected result in the urlbar panel, or -1 for no selection. There won't be a selection when a one-off button is the only selection, and for the `paste_go` or `drop_go` objects. There may also not be a selection if the system was busy and results arrived too late, then we directly decide whether to search or visit the given string without having a fully built result. This is only present for `engagement` events.
- `provider` The name of the result provider for the selected result. Existing values are: `HeuristicFallback`, `Autofill`, `UnifiedComplete`, `TokenAliasEngines`, `SearchSuggestions`, `UrlbarProviderTopSites`. Values can also be defined by URLBar provider experiments.

3.4 Custom pings for Contextual Services

Contextual Services currently has two features running within the Urlbar: TopSites and QuickSuggest. We send various pings as the custom pings to record the impressions and clicks of these two features.

TopSites Impression This records an impression when a sponsored TopSite is shown.

- `context_id` A UUID representing this user. Note that it's not client_id, nor can it be used to link to a client_id.
- `tile_id` A unique identifier for the sponsored TopSite.
- `source` The browser location where the impression was displayed.
- `position` The placement of the TopSite (1-based).
- `advertiser` The Name of the advertiser.
- `reporting_url` The reporting URL of the sponsored TopSite, normally pointing to the ad partner's reporting endpoint.
- `version` Firefox version.
- `release_channel` Firefox release channel.
- `locale` User's current locale.

TopSites Click This records a click ping when a sponsored TopSite is clicked by the user.

- `context_id` A UUID representing this user. Note that it's not client_id, nor can it be used to link to a client_id.
- `tile_id` A unique identifier for the sponsored TopSite.
- `source` The browser location where the click was tirggered.
- `position` The placement of the TopSite (1-based).
- `advertiser` The Name of the advertiser.
- `reporting_url` The reporting URL of the sponsored TopSite, normally pointing to the ad partner's reporting endpoint.
- `version` Firefox version.
- `release_channel` Firefox release channel.
- `locale` User's current locale.

QuickSuggest Impression

This records an impression when the following two conditions hold:

- A user needs to complete the search action by picking a result from the Urlbar
- There must be a QuickSuggest link shown at the end of that search action. No impression will be recorded for any QuickSuggest links that are shown during the user typing, only the last one (if any) counts

Payload:

- `context_id` A UUID representing this user. Note that it's not client_id, nor can it be used to link to a client_id.
- `search_query` The exact search query typed in by the user.
- `matched_keywords` The matched keywords that leads to the QuickSuggest link.
- `is_clicked` Whether or not the use has clicked on the QuickSuggest link.
- `block_id` A unique identifier for a QuickSuggest link (a.k.a a keywords block).
- `position` The placement of the QuickSuggest link in the Urlbar (1-based).
- `advertiser` The Name of the advertiser.
- `reporting_url` The reporting URL of the QuickSuggest link, normally pointing to the ad partner's reporting endpoint.

QuickSuggest Click This records a click ping when a QuickSuggest link is clicked by the user.

- `context_id` A UUID representing this user. Note that it's not client_id, nor can it be used to link to a client_id.
- `advertiser` The Name of the advertiser.
- `block_id` A unique identifier for a QuickSuggest link (a.k.a a keywords block).
- `position` The placement of the QuickSuggest link in the Urlbar (1-based).
- `reporting_url` The reporting URL of the QuickSuggest link, normally pointing to the ad partner's reporting endpoint.

3.5 Other telemetry relevant to the Address Bar

Search Telemetry Some of the search telemetry is also relevant to the address bar.

contextual.services.topsites.* These keyed scalars instrument the impressions and clicks for sponsored TopSites in the urlbar. The key is a combination of the source and the placement of the TopSites link (1-based) such as 'urlbar_1'. For each key, it records the counter of the impression or click. Note that these scalars are shared with the TopSites on the newtab page.

contextual.services.quicksuggest.* These keyed scalars record impressions and clicks on Quick Suggest results, also called Firefox Suggest results, in the address bar. The keys for each scalar are the 1-based indexes of the Quick Suggest results, and the values are the number of impressions or clicks for the corresponding indexes. For example, for a Quick Suggest impression at 0-based index 9, the value for key `10` will be incremented in the `contextual.services.quicksuggest.impression` scalar.

The keyed scalars are:

- `contextual.services.quicksuggest.impression` Incremented when a Quick Suggest result is shown in an address bar engagement where the user picks any result. The particular picked result doesn't matter, and it doesn't need to be the Quick Suggest result.
- `contextual.services.quicksuggest.click` Incremented when the user picks a Quick Suggest result (not including the help button).
- `contextual.services.quicksuggest.help` Incremented when the user picks the onboarding help button in a Quick Suggest result.

contextservices.quicksuggest This is event telemetry under the `contextservices.quicksuggest` category. It's enabled only when the `browser.urlbar.quicksuggest.enabled` pref is true. An event is recorded when the user toggles the `browser.urlbar.suggest.quicksuggest` pref, which corresponds to the checkbox in about:preferences#search labeled "Show Firefox Suggest in the address bar (suggested and sponsored results)". If the user never toggles the pref, then this event is never recorded.

The full spec for this event is:

- Category: `contextservices.quicksuggest`
- Method: `enable_toggled`
- Objects: `enabled`, `disabled` – `enabled` is recorded when the pref is flipped from false to true, and `disabled` is recorded when the pref is flipped from true to false.
- Value: Not used
- Extra: Not used

3.6 Obsolete probes

3.6.1 Obsolete histograms

FX_URLBAR_SELECTED_RESULT_INDEX (OBSOLETE) This probe tracked the indexes of picked results in the results list. It was an enumerated histogram with 17 buckets.

FX_URLBAR_SELECTED_RESULT_TYPE and FX_URLBAR_SELECTED_RESULT_TYPE_2 (from Firefox 78 on) (OBSO This probe tracked the types of picked results. It was an enumerated histogram with 17 buckets:

0. autofill
1. bookmark
2. history
3. keyword
4. searchengine
5. searchsuggestion
6. switchtab
7. tag
8. visiturl
9. remotetab
10. extension
11. preloaded-top-site

12. tip
13. topsite
14. formhistory
15. dynamic
16. tabtosearch

FX_URLBAR_SELECTED_RESULT_INDEX_BY_TYPE and FX_URLBAR_SELECTED_RESULT_INDEX_BY_TYPE_2 (f

This probe tracked picked result type, for each one it tracked the index where it appeared. It was a keyed histogram where the keys were result types (see FX_URLBAR_SELECTED_RESULT_TYPE above). For each key, this recorded the indexes of picked results for that result type.

CHAPTER

FOUR

DEBUGGING & LOGGING

Content to be written

CHAPTER

FIVE

EXTENSIONS & EXPERIMENTS

This document describes address bar extensions and experiments: what they are, how to run them, how to write them, and the processes involved in each.

The primary purpose right now for writing address bar extensions is to run address bar experiments. But extensions are useful outside of experiments, and not all experiments use extensions.

Like all Firefox extensions, address bar extensions use the WebExtensions framework.

5.1 WebExtensions

WebExtensions is the name of Firefox's extension architecture. The "web" part of the name hints at the fact that Firefox extensions are built using Web technologies: JavaScript, HTML, CSS, and to a certain extent the DOM.

Individual extensions themselves often are referred to as *WebExtensions*. For clarity and conciseness, this document will refer to WebExtensions as *extensions*.

Why are we interested in extensions? Mainly because they're a powerful way to run experiments in Firefox. See *Experiments* for more on that. In addition, we'd also like to build up a robust set of APIs useful to extension authors, although right now the API can only be used by Mozilla extensions.

WebExtensions are introduced and discussed in detail on MDN. You'll need a lot of that knowledge in order to build address bar extensions.

5.2 Developing Address Bar Extensions

5.2.1 Overview

The address bar WebExtensions API currently lives in two API namespaces, `browser.urlbar` and `browser.experiments.urlbar`. The reason for this is historical and is discussed in the *Developing Address Bar Extension APIs* section. As a consumer of the API, there are only two important things you need to know:

- There's no meaningful difference between the APIs of the two namespaces. Their kinds of functions, events, and properties are similar. You should think of the address bar API as one single API that happens to be split into two namespaces.
- However, there is a big difference between the two when it comes to setting up your extension to use them. This is discussed next.

The `browser.urlbar` API namespace is built into Firefox. It's a **privileged API**, which means that only Mozilla-signed and temporarily installed extensions can use it. The only thing your Mozilla extension needs to do in order to use it is to request the `urlbar` permission in its manifest.json, as illustrated here.

In contrast, the `browser.experiments.urlbar` API namespace is bundled inside your extension. APIs that are bundled inside extensions are called **experimental APIs**, and the extensions in which they're bundled are called **WebExtension experiments**. As with privileged APIs, experimental APIs are available only to Mozilla-signed and temporarily installed extensions. ("WebExtension experiments" is a term of art and shouldn't be confused with the general notion of experiments that happen to use extensions.) For more on experimental APIs and WebExtension experiments, see the WebExtensions API implementation documentation.

Since `browser.experiments.urlbar` is bundled inside your extension, you'll need to include it in your extension's repo by doing the following:

1. The implementation consists of two files, api.js and schema.json. In your extension repo, create a *experiments/urlbar* subdirectory and copy the files there. See this repo for an example.
2. Add the following `experiment_apis` key to your manifest.json (see here for an example in context):

```
"experiment_apis": {
  "experiments_urlbar": {
    "schema": "experiments/urlbar/schema.json",
    "parent": {
      "scopes": ["addon_parent"],
      "paths": [["experiments", "urlbar"]],
      "script": "experiments/urlbar/api.js"
    }
  }
}
```

As mentioned, only Mozilla-signed and temporarily installed extensions can use these two API namespaces. For information on running the extensions you develop that use these namespaces, see *Running Address Bar Extensions*.

5.2.2 browser.urlbar

Currently the only documentation for `browser.urlbar` is its schema. Fortunately WebExtension schemas are JSON and aren't too hard to read. If you need help understanding it, see the WebExtensions API implementation documentation.

For examples on using the API, see the *Cookbook* section.

5.2.3 browser.experiments.urlbar

As with `browser.urlbar`, currently the only documentation for `browser.experiments.urlbar` is its schema. For examples on using the API, see the *Cookbook* section.

5.2.4 Workflow

The web-ext command-line tool makes the extension-development workflow very simple. Simply start it with the *run* command, passing it the location of the Firefox binary you want to use. web-ext will launch your Firefox and remain running until you stop it, watching for changes you make to your extension's files. When it sees a change, it automatically reloads your extension — in Firefox, in the background — without your having to do anything. It's really nice.

The web-ext documentation lists all its options, but here are some worth calling out for the *run* command:

`--browser-console` Automatically open the browser console when Firefox starts. Very useful for watching your extension's console logging. (Make sure "Show Content Messages" is checked in the console.)

`-p` This option lets you specify a path to a profile directory.

--keep-profile-changes Normally web-ext doesn't save any changes you make to the profile. Use this option along with `-p` to reuse the same profile again and again.

--verbose web-ext suppresses Firefox messages in the terminal unless you pass this option. If you've added some `dump` calls in Firefox because you're working on a new `browser.urlbar` API, for example, you won't see them without this.

web-ext also has a *build* command that packages your extension's files into a zip file. The following *build* options are useful:

--overwrite-dest Without this option, web-ext won't overwrite a zip file it previously created.

web-ext can load its configuration from your extension's package.json. That's the recommended way to configure it. Here's an example.

Finally, web-ext can also sign extensions, but if you're developing your extension for an experiment, you'll use a different process for signing. See *The Experiment Development Process*.

5.2.5 Automated Tests

It's possible to write browser chrome mochitests for your extension the same way we write tests for Firefox. One of the example extensions linked throughout this document includes a test, for instance.

See the readme in the example-addon-experiment repo for a workflow.

5.2.6 Cookbook

To be written. For now, you can find example uses of `browser.experiments.urlbar` and `browser.urlbar` in the following repos:

- https://github.com/mozilla-extensions/firefox-quick-suggest-weather
- https://github.com/0c0w3/urlbar-tips-experiment
- https://github.com/0c0w3/urlbar-top-sites-experiment
- https://github.com/0c0w3/urlbar-search-interventions-experiment

5.2.7 Further Reading

WebExtensions on MDN The place to learn about developing WebExtensions in general.

Getting started with web-ext MDN's tutorial on using web-ext.

web-ext command reference MDN's documentation on web-ext's commands and their options.

5.3 Developing Address Bar Extension APIs

5.3.1 Built-In APIs vs. Experimental APIs

Originally we developed the address bar extension API in the `browser.urlbar` namespace, which is built into Firefox as discussed above. By "built into Firefox," we mean that the API is developed in mozilla-central and shipped inside Firefox just like any other Firefox feature. At the time, that seemed like the right thing to do because we wanted to build an API that ultimately could be used by all extension authors, not only Mozilla.

However, there were a number of disadvantages to this development model. The biggest was that it tightly coupled our experiments to specific versions of Firefox. For example, if we were working on an experiment that targeted Firefox 72, then any APIs used by that experiment needed to land and ship in 72. If we weren't able to finish an API by the time 72 shipped, then the experiment would have to be postponed until 73. Our experiment development timeframes were always very short because we always wanted to ship our experiments ASAP. Often we targeted the Firefox version that was then in Nightly; sometimes we even targeted the version in Beta. Either way, it meant that we were always uplifting patch after patch to Beta. This tight coupling between Firefox versions and experiments erased what should have been a big advantage of implementing experiments as extensions in the first place: the ability to ship experiments outside the usual cyclical release process.

Another notable disadvantage of this model was just the cognitive weight of the idea that we were developing APIs not only for ourselves and our experiments but potentially for all extensions. This meant that not only did we have to design APIs to meet our immediate needs, we also had to imagine use cases that could potentially arise and then design for them as well.

For these reasons, we stopped developing `browser.urlbar` and created the `browser.experiments.urlbar` experimental API. As discussed earlier, experimental APIs are APIs that are bundled inside extensions. Experimental APIs can do anything that built-in APIs can do with the added flexibility of not being tied to specific versions of Firefox.

5.3.2 Adding New APIs

All new address bar APIs should be added to `browser.experiments.urlbar`. Although this API does not ship in Firefox, it's currently developed in mozilla-central, in browser/components/urlbar/tests/ext/ – note the "tests" subdirectory. Developing it in mozilla-central lets us take advantage of our usual build and testing infrastructure. This way we have API tests running against each mozilla-central checkin, against all versions of Firefox that are tested on Mozilla's infrastructure, and we're alerted to any breaking changes we accidentally make. When we start a new extension repo, we copy schema.json and api.js to it as described earlier (or clone an example repo with up-to-date copies of these files).

Generally changes to the API should be reviewed by someone on the address bar team and someone on the WebExtensions team. Shane (mixedpuppy) is a good contact.

5.3.3 Anatomy of an API

Roughly speaking, a WebExtensions API implementation comprises three different pieces:

Schema The schema declares the functions, properties, events, and types that the API makes available to extensions. Schemas are written in JSON.

The `browser.experiments.urlbar` schema is schema.json, and the `browser.urlbar` schema is urlbar.json.

For reference, the schemas of built-in APIs are in browser/components/extensions/schemas and toolkit/components/extensions/schemas.

Internals Every API hooks into some internal part of Firefox. For the address bar API, that's the Urlbar implementation in browser/components/urlbar.

Glue Finally, there's some glue code that implements everything declared in the schema. Essentially, this code mediates between the previous two pieces. It translates the function calls, property accesses, and event listener registrations made by extensions using the public-facing API into terms that the Firefox internals understand, and vice versa.

For `browser.experiments.urlbar`, this is api.js, and for `browser.urlbar`, it's ext-urlbar.js.

For reference, the implementations of built-in APIs are in browser/components/extensions and toolkit/components/extensions, in the *parent* and *child* subdirecties. As you might guess, code in *parent* runs in the main process, and code in *child* runs in the extensions process. Address bar APIs deal with browser chrome and their implementations therefore run in the parent process.

Keep in mind that extensions run in a different process from the main process. That has implications for your APIs. They'll generally need to be async, for example.

5.3.4 Further Reading

WebExtensions API implementation documentation Detailed info on implementing a WebExtensions API.

5.4 Running Address Bar Extensions

As discussed above, `browser.experiments.urlbar` and `browser.urlbar` are privileged APIs. There are two different points to consider when it comes to running an extension that uses privileged APIs: loading the extension in the first place, and granting it access to privileged APIs. There's a certain bar for loading any extension regardless of its API usage that depends on its signed state and the Firefox build you want to run it in. There's yet a higher bar for granting it access to privileged APIs. This section discusses how to load extensions so that they can access privileged APIs.

Since we're interested in extensions primarily for running experiments, there are three particular signed states relevant to us:

Unsigned There are two ways to run unsigned extensions that use privileged APIs.

They can be loaded temporarily using a Firefox Nightly build or Developer Edition but not Beta or Release [source], and the `extensions.experiments.enabled` preference must be set to true [source]. You can load extensions temporarily by visiting about:debugging#/runtime/this-firefox and clicking "Load Temporary Add-on." *web-ext* also loads extensions temporarily.

They can be also be loaded normally (not temporarily) in a custom build where the build-time setting `AppConstants.MOZ_REQUIRE_SIGNING` [source, source] and `xpinstall.signatures.required` pref are both false. As in the previous paragraph, such builds include Nightly and Developer Edition but not Beta or Release [source]. In addition, your custom build must modify the `Extension.isPrivileged` getter to return true. This getter determines whether an extension can access privileged APIs.

Extensions remain unsigned as you develop them. See the *Workflow* section for more.

Signed for testing (Signed for QA) Signed-for-testing extensions that use privileged APIs can be run using the same techniques for running unsigned extensions.

They can also be loaded normally (not temporarily) if you use a Firefox build where the build-time setting `AppConstants.MOZ_REQUIRE_SIGNING` is false and you set the `xpinstall.signatures.dev-root` pref to true [source]. `xpinstall.signatures.dev-root` does not exist by default and must be created.

You encounter extensions that are signed for testing when you are writing extensions for experiments. See the *Experiments* section for details.

"Signed for QA" is another way of referring to this signed state.

Signed for release Signed-for-release extensions that use privileged APIs can be run in any Firefox build with no special requirements.

You encounter extensions that are signed for release when you are writing extensions for experiments. See the *Experiments* section for details.

Important: To see console logs from extensions in the browser console, select the "Show Content Messages" option in the console's settings. This is necessary because extensions run outside the main process.

5.5 Experiments

Experiments let us try out ideas in Firefox outside the usual release cycle and on particular populations of users.

For example, say we have a hunch that the top sites shown on the new-tab page aren't very discoverable, so we want to make them more visible. We have one idea that might work — show them every time the user begins an interaction with the address bar — but we aren't sure how good an idea it is. So we test it. We write an extension that does just that, make sure it collects telemetry that will help us answer our question, ship it outside the usual release cycle to a small percentage of Beta users, collect and analyze the telemetry, and determine whether the experiment was successful. If it was, then we might want to ship the feature to all Firefox users.

Experiments sometimes are also called **studies** (not to be confused with *user studies*, which are face-to-face interviews with users conducted by user researchers).

There are two types of experiments:

Pref-flip experiments Pref-flip experiments are simple. If we have a fully baked feature in the browser that's preffed off, a pref-flip experiment just flips the pref on, enabling the feature for users running the experiment. No code is required. We tell the experiments team the name of the pref we want to flip, and they handle it.

One important caveat to pref-flip studies is that they're currently capable of flipping only a single pref. There's an extension called Multipreffer that can flip multiple prefs, though.

Add-on experiments Add-on experiments are much more complex but much more powerful. (Here *add-on* is a synonym for extension.) They're the type of experiments that this document has been discussing all along.

An add-on experiment is shipped as an extension that we write and that implements the experimental feature we want to test. To reiterate, the extension is a WebExtension and uses WebExtensions APIs. If the current WebExtensions APIs do not meet the needs of your experiment, then you must create either experimental or built-in APIs so that your extension can use them. If necessary, you can make any new built-in APIs privileged so that they are available only to Mozilla extensions.

An add-on experiment can collect additional telemetry that's not collected in the product by using the privileged `browser.telemetry` WebExtensions API, and of course the product will continue to collect all the telemetry it usually does. The telemetry pings from users running the experiment will be correlated with the experiment with no extra work on our part.

A single experiment can deliver different UXes to different groups of users running the experiment. Each group or UX within an experiment is called a **branch**. Experiments often have two branches, control and treatment. The **control branch** actually makes no UX changes. It may capture additional telemetry, though. Think of it as the control in a science experiment. It's there so we can compare it to data from the **treatment branch**, which does make UX changes. Some experiments may require multiple treatment branches, in which case the different branches will have different names. Add-on experiments can implement all branches in the same extension or each branch in its own extension.

Experiments are delivered to users by a system called **Normandy**. Normandy comprises a client side that lives in Firefox and a server side. In Normandy, experiments are defined server-side in files called **recipes**. Recipes include information about the experiment like the Firefox release channel and version that the experiment targets, the number of users to be included in the experiment, the branches in the experiment, the percentage of users on each branch, and so on.

Experiments are tracked by Mozilla project management using a system called Experimenter.

Finally, there was an older version of the experiments program called **Shield**. Experiments under this system were called **Shield studies** and could be be shipped as extensions too.

5.5.1 Further Reading

Pref-Flip and Add-On Experiments A comprehensive document on experiments from the Experimenter team. See the child pages in the sidebar, too.

Client Implementation Guidelines for Experiments Relevant documentation from the telemetry team.

#ask-experimenter Slack channel A friendly place to get answers to your experiment questions.

5.6 The Experiment Development Process

This section describes an experiment's life cycle.

1. Experiments usually originate with product management and UX. They're responsible for identifying a problem, deciding how an experiment should approach it, the questions we want to answer, the data we need to answer those questions, the user population that should be enrolled in the experiment, the definition of success, and so on.
2. UX makes a spec that describes what the extension looks like and how it behaves.
3. There's a kickoff meeting among the team to introduce the experiment and UX spec. It's an opportunity for engineering to ask questions of management, UX, and data science. It's really important for engineering to get a precise and accurate understanding of how the extension is supposed to behave — right down to the UI changes — so that no one makes erroneous assumptions during development.
4. At some point around this time, the team (usually management) creates a few artifacts to track the work and facilitate communication with outside teams involved in shipping experiments. They include:
   - A page on *Experimenter*
   - A QA PI (product integrity) request so that QA resources are allocated
   - A bug in Data Science :: Experiment Collaboration so that data science can track the work and discuss telemetry (engineering might file this one)
5. Engineering breaks down the work and files bugs. There's another engineering meeting to discuss the breakdown, or it's discussed asynchronously.
6. Engineering sets up a GitHub repo for the extension. See *Implementing Experiments* for an example repo you can clone to get started. Disable GitHub Issues on the repo so that QA will file bugs in Bugzilla instead of GitHub. There's nothing wrong with GitHub Issues, but our team's project management tracks all work through Bugzilla. If it's not there, it's not captured.
7. Engineering or management fills out the Add-on section of the Experimenter page as much as possible at this point. "Active Experiment Name" isn't necessary, and "Signed Release URL" won't be available until the end of the process.
8. Engineering implements the extension and any new WebExtensions APIs it requires.
9. When the extension is done, engineering or management clicks the "Ready for Sign-Off" button on the Experimenter page. That changes the page's status from "Draft" to "Ready for Sign-Off," which allows QA and other teams to sign off on their portions of the experiment.
10. Engineering requests the extension be signed "for testing" (or "for QA"). Michael (mythmon) from the Experiments team and Rehan (rdalal) from Services Engineering are good contacts. Build the extension zip file using web-ext as discussed in *Workflow*. Attach it to a bug (a metabug for implementing the extension, for example),

needinfo Michael or Rehan, and ask him to sign it. He'll attach the signed version to the bug. If neither Michael nor Rehan is available, try asking in the #ask-experimenter Slack channel.

11. Engineering sends QA the link to the signed extension and works with them to resolve bugs they find.
12. When QA signs off, engineering asks Michael to sign the extension "for release" using the same needinfo process described earlier.
13. Paste the URL of the signed extension in the "Signed Release URL" textbox of the Add-on section of the Experimenter page.
14. Other teams sign off as they're ready.
15. The experiment ships!

5.7 Implementing Experiments

This section discusses how to implement add-on experiments. Pref-flip experiments are much simpler and don't need a lot of explanation. You should be familiar with the concepts discussed in the *Developing Address Bar Extensions* and *Running Address Bar Extensions* sections before reading this one.

The most salient thing about add-on experiments is that they're implemented simply as privileged extensions. Other than being privileged and possibly containing bundled experimental APIs, they're similar to all other extensions.

The top-sites experiment extension is an example of a real, shipped experiment.

5.7.1 Setup

example-addon-experiment is a repo you can clone to get started. It's geared toward urlbar extensions and includes the stub of a browser chrome mochitest.

5.7.2 browser.normandyAddonStudy

As discussed in *Experiments*, an experiment typically has more than one branch so that it can test different UXes. The experiment's extension(s) needs to know the branch the user is enrolled in so that it can behave appropriately for the branch: show the user the proper UX, collect the proper telemetry, and so on.

This is the purpose of the `browser.normandyAddonStudy` WebExtensions API. Like `browser.urlbar`, it's a privileged API available only to Mozilla extensions.

Its schema is normandyAddonStudy.json.

It's a very simple API. The primary function is `getStudy`, which returns the study the user is currently enrolled in or null if there isn't one. (Recall that *study* is a synonym for *experiment*.) One of the first things an experiment extension typically does is to call this function.

The Normandy client in Firefox will keep an experiment extension installed only while the experiment is active. Therefore, `getStudy` should always return a non-null study object. Nevertheless, the study object has an `active` boolean property that's trivial to sanity check. (The example extension does.)

The more important property is `branch`, the name of the branch that the user is enrolled in. Your extension should use it to determine the appropriate UX.

Finally, there's an `onUnenroll` event that's fired when the user is unenrolled in the study. It's not quite clear in what cases an extension would need to listen for this event given that Normandy automatically uninstalls extensions on unenrollment. Maybe if they create some persistent state that's not automatically undone on uninstall by the WebExtensions framework?

If your extension itself needs to unenroll the user for some reason, call `endStudy`.

5.7.3 Telemetry

Experiments can capture telemetry in two places: in the product itself and through the privileged `browser.telemetry` WebExtensions API. The API schema is telemetry.json.

The telemetry pings from users running experiments are automatically correlated with those experiments, no extra work required. That's true regardless of whether the telemetry is captured in the product or though `browser.telemetry`.

The address bar has some in-product, preffed off telemetry that we want to enable for all our experiments — at least that's the thinking as of August 2019. It's called engagement event telemetry, and it records user *engagements* with and *abandonments* of the address bar [source]. We added a BrowserSetting on `browser.urlbar` just to let us flip the pref and enable this telemetry in our experiment extensions. Call it like this:

```
await browser.urlbar.engagementTelemetry.set({ value: true });
```

5.7.4 Engineering Best Practices

Clear up questions with your UX person early and often. There's often a gap between what they have in their mind and what you have in yours. Nothing wrong with that, it's just the nature of development. But misunderstandings can cause big problems when they're discovered late. This is especially true of UX behaviors, as opposed to visuals or styling. It's no fun to realize at the end of a release cycle that you've designed the wrong WebExtensions API because some UX detail was overlooked.

Related to the previous point, make builds of your extension for your UX person so they can test it.

Taking the previous point even further, if your experiment will require a substantial new API(s), you might think about prototyping the experiment entirely in a custom Firefox build before designing the API at all. Give it to your UX person. Let them disect it and tell you all the problems with it. Fill in all the gaps in your understanding, and then design the API. We've never actually done this, though.

It's a good idea to work on the extension as you're designing and developing the APIs it'll use. You might even go as far as writing the first draft of the extension before even starting to implement the APIs. That lets you spot problems that may not be obvious were you to design the API in isolation.

Your extension's ID should end in `@shield.mozilla.org`. QA will flag it if it doesn't.

Set `"hidden": true` in your extension's manifest.json. That hides it on about:addons. (It can still be seen on about:studies.) QA will spot this if you don't.

There are drawbacks of hiding features behind prefs and enabling them in experiment extensions. Consider not doing that if feasible, or at least weigh these drawbacks against your expected benefits.

- Prefs stay flipped on in private windows, but experiments often have special requirements around private-browsing mode (PBM). Usually, they shouldn't be active in PBM at all, unless of course the point of the experiment is to test PBM. Extensions also must request PBM access ("incognito" in WebExtensions terms), and the user can disable access at any time. The result is that part of your experiment could remain enabled — the part behind the pref — while other parts are disabled.
- Prefs stay flipped on in safe mode, even though your extension (like all extensions) will be disabled. This might be a bug in the WebExtensions framework, though.

CHAPTER

SIX

DYNAMIC RESULT TYPES

This document discusses a special category of address bar results called dynamic result types. Dynamic result types allow you to easily add new types of results to the address bar and are especially useful for extensions.

The intended audience for this document is developers who need to add new kinds of address bar results, either internally in the address bar codebase or through extensions.

6.1 Motivation

The address bar provides many different types of results in normal Firefox usage. For example, when you type a search term, the address bar may show you search suggestion results from your current search engine. It may also show you results from your browsing history that match your search. If you typed a certain phrase like "update Firefox," it will show you a tip result that lets you know whether Firefox is up to date.

Each of these types of results is built into the address bar implementation. If you wanted to add a new type of result – say, a card that shows the weather forecast when the user types "weather" – one way to do so would be to add a new result type. You would need to update all the code paths in the address bar that relate to result types. For instance, you'd need to update the code path that handles clicks on results so that your weather card opens an appropriate forecast URL when clicked; you'd need to update the address bar view (the panel) so that your card is drawn correctly; you may need to update the keyboard selection behavior if your card contains elements that can be independently selected such as different days of the week; and so on.

If you're implementing your weather card in an extension, as you might in an add-on experiment, then you'd need to land your new result type in mozilla-central so your extension can use it. Your new result type would ship with Firefox even though the vast majority of users would never see it, and your fellow address bar hackers would have to work around your code even though it would remain inactive most of the time, at least until your experiment graduated.

6.2 Dynamic Result Types

Dynamic result types are an alternative way of implementing new result types. Instead of adding a new built-in type along with all that entails, you add a new provider subclass and register a template that describes how the view should draw your result type and indicates which elements are selectable. The address bar takes care of everything else. (Or if you're implementing an extension, you add a few event handlers instead of a provider subclass, although we have a shim that abstracts away the differences between internal and extension address bar code.)

Dynamic result types are essentially an abstraction layer: Support for them as a general category of results is built into the address bar, and each implementation of a specific dynamic result type fills in the details.

In addition, dynamic result types can be added at runtime. This is important for extensions that implement new types of results like the weather forecast example above.

6.3 Getting Started

To get a feel for how dynamic result types are implemented, you can look at the example dynamic result type extension. The extension uses the recommended shim that makes writing address bar extension code very similar to writing internal address bar code, and it's therefore a useful example even if you intend to add a new dynamic result type internally in the address bar codebase in mozilla-central.

The next section describes the specific steps you need to take to add a new dynamic result type.

6.4 Implementation Steps

This section describes how to add a new dynamic result type in either of the following cases:

- You want to add a new dynamic result type in an extension using the recommended shim.
- You want to add a new dynamic result type internal to the address bar codebase in mozilla-central.

The steps are mostly the same in both cases and are described next.

If you want to add a new dynamic result type in an extension but don't want to use the shim, then skip ahead to *Appendix B: Using the WebExtensions API Directly*.

6.4.1 1. Register the dynamic result type

First, register the new dynamic result type:

```
UrlbarResult.addDynamicResultType(name);
```

`name` is a string identifier for the new type. It must be unique; that is, it must be different from all other dynamic result type names. It will also be used in DOM IDs, DOM class names, and CSS selectors, so it should not contain any spaces or other characters that are invalid in CSS.

6.4.2 2. Register the view template

Next, add the view template for the new type:

```
UrlbarView.addDynamicViewTemplate(name, viewTemplate);
```

`name` is the new type's name as described in step 1.

`viewTemplate` is an object called a view template. It describes in a declarative manner the DOM that should be created in the view for all results of the new type. For providers created in extensions, it also declares the stylesheet that should be applied to results in the view. See *View Templates* for a description of this object.

6.4.3 3. Add the provider

As with any type of result, results for dynamic result types must be created by one or more providers. Make a `UrlbarProvider` subclass for the new provider and implement all the usual provider methods as you normally would:

```
class MyDynamicResultTypeProvider extends UrlbarProvider {
  // ...
}
```

The `startQuery` method should create `UrlbarResult` objects with the following two requirements:

- Result types must be `UrlbarUtils.RESULT_TYPE.DYNAMIC`.
- Result payloads must have a `dynamicType` property whose value is the name of the dynamic result type used in step 1.

The results' sources, other payload properties, and other result properties aren't relevant to dynamic result types, and you should choose values appropriate to your use case.

If any elements created in the view for your results can be picked with the keyboard or mouse, then be sure to implement your provider's `pickResult` method.

For help on implementing providers in general, see the address bar's Architecture Overview.

If you are creating the provider in the internal address bar implementation in mozilla-central, then don't forget to register it in `UrlbarProvidersManager`.

If you are creating the provider in an extension, then it's registered automatically, and there's nothing else you need to do.

6.4.4 4. Implement the provider's getViewUpdate method

`getViewUpdate` is a provider method particular to dynamic result type providers. Its job is to update the view DOM for a specific result. It's called by the view for each result in the view that was created by the provider. It returns an object called a view update object.

Recall that the view template was added earlier, in step 2. The view template describes how to build the DOM structure for all results of the dynamic result type. The view update object, in this step, describes how to fill in that structure for a specific result.

Add the `getViewUpdate` method to the provider:

```
/**
 * Returns a view update object that describes how to update the view DOM
 * for a given result.
 *
 * @param {UrlbarResult} result
 *   The view update object describes how to update the view DOM for this
 *   particular result.
 * @param {Map} idsByName
 *   A map from names in the view template to the IDs of their corresponding
 *   elements in the DOM.
 */
getViewUpdate(result, idsByName) {
  let viewUpdate = {
    // ...
  };
```

(continues on next page)

(continued from previous page)

```
  return viewUpdate;
}
```

`result` is the result from the provider for which the view update is being requested.

`idsByName` is a map from names in the view template to the IDs of their corresponding elements in the DOM. This is useful if parts of the view update depend on element IDs, as some ARIA attributes do.

The return value is a view update object. It describes in a declarative manner the updates that should be performed on the view DOM. See *View Update Objects* for a description of this object.

6.4.5 5. Style the results

If you are creating the provider in the internal address bar implementation in mozilla-central, then add styling dynamicResults.inc.css.

If you are creating the provider in an extension, then bundle a CSS file in your extension and declare it in the top-level `stylesheet` property of your view template, as described in *View Templates*. Additionally, if any of your rules override built-in rules, then you'll need to declare them as `!important`.

The rest of this section will discuss the CSS rules you need to use to style your results.

There are two DOM annotations that are useful for styling. The first is the `dynamicType` attribute that is set on result rows, and the second is a class that is set on child elements created from the view template.

dynamicType Row Attribute

The topmost element in the view corresponding to a result is called a **row**. Rows have a class of `urlbarView-row`, and rows corresponding to results of a dynamic result type have an attributed called `dynamicType`. The value of this attribute is the name of the dynamic result type that was chosen in step 1 earlier.

Rows of a specific dynamic result type can therefore be selected with the following CSS selector, where `TYPE_NAME` is the name of the type:

```
.urlbarView-row[dynamicType=TYPE_NAME]
```

Child Element Class

As discussed in *View Templates*, each object in the view template can have a `name` property. The elements in the view corresponding to the objects in the view template receive a class named `urlbarView-dynamic-TYPE_NAME-ELEMENT_NAME`, where `TYPE_NAME` is the name of the dynamic result type, and `ELEMENT_NAME` is the name of the object in the view template.

Elements in dynamic result type rows can therefore be selected with the following:

```
.urlbarView-dynamic-TYPE_NAME-ELEMENT_NAME
```

If an object in the view template does not have a `name` property, then it won't receive the class and it therefore can't be selected using this selector.

6.5 View Templates

A **view template** is a plain JS object that declaratively describes how to build the DOM for a dynamic result type. When a result of a particular dynamic result type is shown in the view, the type's view template is used to construct the part of the view that represents the type in general.

The need for view templates arises from the fact that extensions run in a separate process from the chrome process and can't directly access the chrome DOM, where the address bar view lives. Since extensions are a primary use case for dynamic result types, this is an important constraint on their design.

6.5.1 Properties

A view template object is a tree-like nested structure where each object in the nesting represents a DOM element to be created. This tree-like structure is achieved using the `children` property described below. Each object in the structure may include the following properties:

`{string} name` The name of the object. This is required for all objects in the structure except the root object and serves two important functions:

1. The element created for the object will automatically have a class named `urlbarView-dynamic-${dynamicType}-${name}`, where `dynamicType` is the name of the dynamic result type. The element will also automatically have an attribute `name` whose value is this name. The class and attribute allow the element to be styled in CSS.
2. The name is used when updating the view, as described in *View Update Objects*.

Names must be unique within a view template, but they don't need to be globally unique. In other words, two different view templates can use the same names, and other unrelated DOM elements can use the same names in their IDs and classes.

`{string} tag` The element tag name of the object. This is required for all objects in the structure except the root object and declares the kind of element that will be created for the object: `span`, `div`, `img`, etc.

`{object} [attributes]` An optional mapping from attribute names to values. For each name-value pair, an attribute is set on the element created for the object.

A special `selectable` attribute tells the view that the element is selectable with the keyboard. The element will automatically participate in the view's keyboard selection behavior.

Similarly, the `role=button` ARIA attribute will also automatically allow the element to participate in keyboard selection. The `selectable` attribute is not necessary when `role=button` is specified.

`{array} [children]` An optional list of children. Each item in the array must be an object as described in this section. For each item, a child element as described by the item is created and added to the element created for the parent object.

`{array} [classList]` An optional list of classes. Each class will be added to the element created for the object by calling `element.classList.add()`.

`{string} [stylesheet]` For dynamic result types created in extensions, this property should be set on the root object in the view template structure, and its value should be a stylesheet URL. The stylesheet will be loaded in all browser windows so that the dynamic result type view may be styled. The specified URL will be resolved against the extension's base URI. We recommend specifying a URL relative to your extension's base directory.

For dynamic result types created internally in the address bar codebase, this value should not be specified and instead styling should be added to dynamicResults.inc.css.

6.5.2 Example

Let's return to the weather forecast example from *earlier*. For each result of our weather forecast dynamic result type, we might want to display a label for a city name along with two buttons for today's and tomorrow's forecasted high and low temperatures. The view template might look like this:

```
{
  stylesheet: "style.css",
  children: [
    {
      name: "cityLabel",
      tag: "span",
    },
    {
      name: "today",
      tag: "div",
      classList: ["day"],
      attributes: {
        selectable: "true",
      },
      children: [
        {
          name: "todayLabel",
          tag: "span",
          classList: ["dayLabel"],
        },
        {
          name: "todayLow",
          tag: "span",
          classList: ["temperature", "temperatureLow"],
        },
        {
          name: "todayHigh",
          tag: "span",
          classList: ["temperature", "temperatureHigh"],
        },
      },
    },
    {
      name: "tomorrow",
      tag: "div",
      classList: ["day"],
      attributes: {
        selectable: "true",
      },
      children: [
        {
          name: "tomorrowLabel",
          tag: "span",
          classList: ["dayLabel"],
        },
        {
          name: "tomorrowLow",
          tag: "span",
          classList: ["temperature", "temperatureLow"],
        },
        {
          name: "tomorrowHigh",
```

(continues on next page)

(continued from previous page)

```
          tag: "span",
          classList: ["temperature", "temperatureHigh"],
        },
      },
    },
  ],
}
```

Observe that we set the special `selectable` attribute on the `today` and `tomorrow` elements so they can be selected with the keyboard.

6.6 View Update Objects

A **view update object** is a plain JS object that declaratively describes how to update the DOM for a specific result of a dynamic result type. When a result of a dynamic result type is shown in the view, a view update object is requested from the result's provider and is used to update the DOM for that result.

Note the difference between view update objects, described in this section, and view templates, described in the previous section. View templates are used to build a general DOM structure appropriate for all results of a particular dynamic result type. View update objects are used to fill in that structure for a specific result.

When a result is shown in the view, first the view looks up the view template of the result's dynamic result type. It uses the view template to build a DOM subtree. Next, the view requests a view update object for the result from its provider. The view update object tells the view which result-specific attributes to set on which elements, result-specific text content to set on elements, and so on. View update objects cannot create new elements or otherwise modify the structure of the result's DOM subtree.

Typically the view update object is based on the result's payload.

6.6.1 Properties

The view update object is a nested structure with two levels. It looks like this:

```
{
  name1: {
    // individual update object for name1
  },
  name2: {
    // individual update object for name2
  },
  name3: {
    // individual update object for name3
  },
  // ...
}
```

The top level maps object names from the view template to individual update objects. The individual update objects tell the view how to update the elements with the specified names. If a particular element doesn't need to be updated, then it doesn't need an entry in the view update object.

Each individual update object can have the following properties:

{object} [attributes] A mapping from attribute names to values. Each name-value pair results in an attribute being set on the element.

{object} [style] A plain object that can be used to add inline styles to the element, like `display: none`. `element.style` is updated for each name-value pair in this object.

{object} [l10n] An `{ id, args }` object that will be passed to `document.l10n.setAttributes()`.

{string} [textContent] A string that will be set as `element.textContent`.

6.6.2 Example

Continuing our weather forecast example, the view update object needs to update several things that we declared in our view template:

- The city label
- The "today" label
- Today's low and high temperatures
- The "tomorrow" label
- Tomorrow's low and high temperatures

Typically, each of these, with the possible exceptions of the "today" and "tomorrow" labels, would come from our results' payloads. There's an important connection between what's in the view and what's in the payloads: The data in the payloads serves the information shown in the view.

Our view update object would then look something like this:

```
{
  cityLabel: {
    textContent: result.payload.city,
  },
  todayLabel: {
    textContent: "Today",
  },
  todayLow: {
    textContent: result.payload.todayLow,
  },
  todayHigh: {
    textContent: result.payload.todayHigh,
  },
  tomorrowLabel: {
    textContent: "Tomorrow",
  },
  tomorrowLow: {
    textContent: result.payload.tomorrowLow,
  },
  tomorrowHigh: {
    textContent: result.payload.tomorrowHigh,
  },
}
```

6.7 Accessibility

Just like built-in types, dynamic result types support a11y in the view, and you should make sure your view implementation is fully accessible.

Since the views for dynamic result types are implemented using view templates and view update objects, in practice supporting a11y for dynamic result types means including appropriate ARIA attributes in the view template and view update objects, using the `attributes` property.

Many ARIA attributes depend on element IDs, and that's why the `idsByName` parameter to the `getViewUpdate` provider method is useful.

Usually, accessible address bar results require the ARIA attribute `role=group` on their top-level DOM element to indicate that all the child elements in the result's DOM subtree form a logical group. This attribute can be set on the root object in the view template.

6.7.1 Example

Continuing the weather forecast example, we'd like for screen readers to know that our result is labeled by the city label so that they announce the city when the result is selected.

The relevant ARIA attribute is `aria-labelledby`, and its value is the ID of the element with the label. In our `getViewUpdate` implementation, we can use the `idsByName` map to get the element ID that the view created for our city label, like this:

```
getViewUpdate(result, idsByName) {
  return {
    root: {
      attributes: {
        "aria-labelledby": idsByName.get("cityLabel"),
      },
    },
    // *snipping the view update object example from earlier*
  };
}
```

Here we're using the name "root" to refer to the root object in the view template, so we also need to update our view template by adding the `name` property to the top-level object, like this:

```
{
  stylesheet: "style.css",
  name: "root",
  attributes: {
    role: "group",
  },
  children: [
    {
      name: "cityLabel",
      tag: "span",
    },
    // *snipping the view template example from earlier*
  ],
}
```

Note that we've also included the `role=group` ARIA attribute on the root, as discussed above. We could have included it in the view update object instead of the view template, but since it doesn't depend on a specific result or element ID in the `idsByName` map, the view template makes more sense.

6.8 Mimicking Built-in Address Bar Results

Sometimes it's desirable to create a new result type that looks and behaves like the usual built-in address bar results. Two conveniences are available that are useful in this case.

6.8.1 URL Navigation

If a result's payload includes a string `url` property and a boolean `shouldNavigate: true` property, then picking the result will navigate to the URL. The `pickResult` method of the result's provider will still be called before navigation.

6.8.2 Text Highlighting

Most built-in address bar results emphasize occurrences of the user's search string in their text by boldfacing matching substrings. Search suggestion results do the opposite by emphasizing the portion of the suggestion that the user has not yet typed. This emphasis feature is called **highlighting**, and it's also available to the results of dynamic result types.

Highlighting for dynamic result types is a fairly automated process. The text that you want to highlight must be present as a property in your result payload. Instead of setting the property to a string value as you normally would, set it to an array with two elements, where the first element is the text and the second element is a `UrlbarUtils.HIGHLIGHT` value, like the `title` payload property in the following example:

```
let result = new UrlbarResult(
  UrlbarUtils.RESULT_TYPE.DYNAMIC,
  UrlbarUtils.RESULT_SOURCE.OTHER_NETWORK,
  {
    title: [
      "Some result title",
      UrlbarUtils.HIGHLIGHT.TYPED,
    ],
    // *more payload properties*
  }
);
```

`UrlbarUtils.HIGHLIGHT` is defined in the extensions shim and is described below.

Your view template must create an element corresponding to the payload property. That is, it must include an object where the value of the `name` property is the name of the payload property, like this:

```
{
  children: [
    {
      name: "title",
      tag: "span",
    },
    // ...
  ],
}
```

In contrast, your view update objects must *not* include an update for the element. That is, they must not include a property whose name is the name of the payload property.

Instead, when the view is ready to update the DOM of your results, it will automatically find the elements corresponding to the payload property, set their `textContent` to the text value in the array, and apply the appropriate highlighting, as described next.

There are two possible `UrlbarUtils.HIGHLIGHT` values. Each controls how highlighting is performed:

`UrlbarUtils.HIGHLIGHT.TYPED` Substrings in the payload text that match the user's search string will be emphasized.

`UrlbarUtils.HIGHLIGHT.SUGGESTED` If the user's search string appears in the payload text, then the remainder of the text following the matching substring will be emphasized.

6.9 Appendix A: Examples

This section lists some example and real-world consumers of dynamic result types.

Example Extension This extension demonstrates a simple use of dynamic result types.

Weather Quick Suggest Extension A real-world Firefox extension experiment that shows weather forecasts and alerts when the user performs relevant searches in the address bar.

Tab-to-Search Provider This is a built-in provider in mozilla-central that uses dynamic result types.

6.10 Appendix B: Using the WebExtensions API Directly

If you're developing an extension, the recommended way of using dynamic result types is to use the shim, which abstracts away the differences between writing internal address bar code and extensions code. The *implementation steps* above apply to extensions as long as you're using the shim.

For completeness, in this section we'll document the WebExtensions APIs that the shim is built on. If you don't use the shim for some reason, then follow these steps instead. You'll see that each step above using the shim has an analogous step here.

The WebExtensions API schema is declared in schema.json and implemented in api.js.

6.10.1 1. Register the dynamic result type

First, register the new dynamic result type:

```
browser.experiments.urlbar.addDynamicResultType(name, type);
```

`name` is a string identifier for the new type. See step 1 in *Implementation Steps* for a description, which applies here, too.

`type` is an object with metadata for the new type. Currently no metadata is supported, so this should be an empty object, which is the default value.

6.10.2 2. Register the view template

Next, add the view template for the new type:

```
browser.experiments.urlbar.addDynamicViewTemplate(name, viewTemplate);
```

See step 2 above for a description of the parameters.

6.10.3 3. Add WebExtension event listeners

Add all the WebExtension event listeners you normally would in an address bar extension, including the two required listeners, `onBehaviorRequested` and and `onResultsRequested`.

```
browser.urlbar.onBehaviorRequested.addListener(query => {
  return "active";
}, providerName);

browser.urlbar.onResultsRequested.addListener(query => {
  let results = [
    // ...
  ];
  return results;
}, providerName);
```

See the address bar extensions document for help on the urlbar WebExtensions API.

6.10.4 4. Add an onViewUpdateRequested event listener

`onViewUpdateRequested` is a WebExtensions event particular to dynamic result types. It's analogous to the `getViewUpdate` provider method described earlier.

```
browser.experiments.urlbar.onViewUpdateRequested.addListener((payload, idsByName) => {
  let viewUpdate = {
    // ...
  };
  return viewUpdate;
});
```

Note that unlike `getViewUpdate`, here the listener's first parameter is a result payload, not the result itself.

The listener should return a view update object.

6.10.5 5. Style the results

This step is the same as step 5 above. Bundle a CSS file in your extension and declare it in the top-level `stylesheet` property of your view template.

CHAPTER

SEVEN

GETTING IN TOUCH

For any questions regarding the Address Bar, the team is available through the #search channel on Slack and the fx-search@mozilla.com mailing list.

Issues can be filed in Bugzilla under the Firefox / Address Bar component.