Omnibox Docs Part 1: Overview

pkasting@, March 24, 2017

Intro

This is part 1 of a series of documents on the Chrome omnibox.

1. Part 1: Overview [this doc]
2. Part 2: UI Classes
3. Part 3: Autocomplete Classes

This doc discusses the core design principles of the omnibox, practical ramifications of those principles, a brief overview of how the system fits together, and a history of some of the more important changes. It also provides links to some older documents. Parts 2 and 3 are more detailed looks at the implementation of the “frontend UI” and “backing system to produce suggestions”, respectively.

Omnibox Design Philosophy

Goals And Assumptions

Overarching Goals

The primary design goal of the omnibox is to reduce users’ cognitive load. Combining search and address inputs into one control eliminates a blocking decision the user would otherwise have to make to begin each such interaction. Stability and predictability of the results make the omnibox trustworthy, and allow users to develop interaction patterns where the same few keystrokes can always be used to access common sites. A sparse presentation in the dropdown avoids distractions while the user types.

In keeping with Chrome’s “Four Ss”, the omnibox should feel speedy and simple, rather than complex and powerful. It should give users exactly what they want as the default result, with the minimum of effort, rather than providing complex operators and modes and asking users to do sequential tasks or peer through dozens of similar results.

From an interaction perspective, the omnibox was designed to accelerate user typing and get users where they want to go quickly, rather than to meet users’ ultimate information needs directly. Meeting those needs is the job of the websites users load, not the omnibox they use to load them. This is similar in philosophy to old Google SERPs, which sought to bounce users away to their desired website as quickly as possible, rather than capture user attention. Over

time, this principle has been challenged; for example, Answers in Suggest clearly violates it. Because such violations can hinder the “simple, sparse, low cognitive load” goals above, they must be evaluated carefully against those goals.

Predictable vs. Magic

User interfaces can often be characterized as either “predictable” or “magic”: one type behaves in a straightforward, easily-explainable way under explicit user control, while the other has behavior that defies explanation but (ideally) solves user needs automatically. “Magic” interfaces tend to delight users when they work well, but infuriate them when things go wrong, because users cannot understand the interaction model enough to recover from bad states or predict (and avoid) them in the future. Such interfaces generally feel untrustworthy, and “predictable” interfaces, where users can mentally model the interaction with high accuracy, are considered safer.

The omnibox tries to be a “magic” interface that still feels “predictable”, and thus gain the benefits of both worlds. Even users who cannot explain how the omnibox works should not be surprised by its behavior. This is accomplished via a few specific principles:

- **No race conditions.** Inline autocompletions and the default behavior on hitting <enter> should never change depending on how fast a user hits the next keystroke. Typing should not be thrown away just because the underlying page navigated during interaction. The dropdown should never change while arrowing through it.
- **Obey the user.** Users should never have to delete an inline autocompletion just to navigate to the exact string they typed. Users should be able to know for certain what hitting <enter> will do before they hit it.
- **Stability within interactions.** If the inline autocompletion is for the site the user wants, continuing to type more of that same input string shouldn’t result in changing away to a different autocompletion. Rankings within the dropdown should not shift around during typing if the results themselves aren’t changing.
- **Stability across interactions.** If a series of keystrokes produce a particular default result once, they should produce the same result next time, unless the user has good reason to expect them not to. A site should be eligible for inline autocompletion based on a stable heuristic (“you’ve typed this before”) rather than a complicated scoring algorithm.

Sometimes these principles conflict. Consider a user who frequently navigates to “moma.org” and occasionally to “moma”. An input like [mo] should probably autocomplete to “moma.org”. But what if the user keeps typing [moma]? “Stability within interactions” argues for not changing the inline autocompletion; “obey the user” argues for removing inline autocompletion so hitting enter will navigate to the intranet site. In such situations, it’s helpful to ask what users would expect to happen if they typed a particular input sequence and hit <enter>, without ever looking at the omnibox. Doing that will generally minimize cognitive load.

Behavior Assumptions

The original, desktop-focused omnibox design makes several assumptions of user behavior. Except for the first bullet, these assumptions refer exclusively to interactions using the omnibox.

- Users may use a mixture of different navigation methods in general (bookmarks, NTP tiles, links from a custom homepage, SERP links, the omnibox). But for each site they revisit, they tend to converge on a common method, and always use that method to access that site in the future.
- Users navigate more than searching.
- Most navigations are to a small set of commonly-visited sites.
- Most such sites have unique hostnames, and users navigate to them by typing them from the beginning of the hostname.
- Users generally don't care about the difference between http:// and https://, or "www." versus no "www.", so omitting them provides little signal of intent. Users who do bother to type them are doing so for a reason.
- Most searches are short (one or two words).
- Most searches are unique (within this user's history).

In practice, these assumptions have been largely correct, but important exceptions have emerged:

- "Users navigate more than searching" may be less true than it used to be, and is likely less true on mobile.
- There are at least two types of sites (uncommonly visited sites, and sites whose URLs are garbage, e.g. individual Google Docs) where users often navigate by typing unique parts of the URL or title instead of the beginning of the hostname.

Practical Ramifications

Inline Autocompletion

Besides "combine search and navigation functionality", inline autocompletion is the central feature of the omnibox. It is what achieves most of the design goals; it can do so because of the behavioral assumptions above; its centrality is why it appears so frequently in the details of the "predictability" principles; and its needs dictated the early code architecture.

Successful inline autocompletion dramatically reduces user typing. Then, by reducing navigations to a few keystrokes, it makes them amenable to muscle memory, driving repeated future usage. Along the way, it displays information about what will happen on hitting enter, so the results of these memorized sequences are predictable. This also eliminates the need for

users to look at or select results in the dropdown, minimizing distraction, eliminating another keystroke, and further increasing interaction speed.

Inline autocompletion only works consistently if users are mostly typing autocompletable strings (i.e. prefixes of a single longer input such as a hostname). It's only helpful when these inputs are uniquely identified by their first few letters, and these unique past inputs do a good job of predicting the user's current desires.

The early omnibox was purpose-built to provide inline autocompletion while remaining predictable. Dropdown suggestions could be returned synchronously or asynchronously in response to user typing, but the scoring model was carefully tuned so asynchronous results would never outscore the top synchronous one; this ensured inline autocompletion wouldn't race hitting the enter key (and make the behavior unpredictable), a problem for every other browser's implementation at the time. The scoring model was based on tables of fixed values rather than more flexible algorithms, because the tables could be tweaked to ensure consistent inline autocompletion within and across interactions. Significant time was spent on inline autocompletion-related edge cases (e.g. "what should <ctrl>-<enter> do if inline autocompletion is present") to ensure the feature felt consistent and trustworthy.

(Some of the early designs have since been improved. The fragile invariant that "asynchronous results must score lower than the top synchronous one" has been replaced by simply marking individual suggestions as suitable or unsuitable to be the default match. Several different suggestion sources now use more algorithmic scoring mechanisms, rather than simple fixed tables, and then tweak their results at the end to maintain any desired invariants.)

Making inline autocompletion the central UX feature, coupled with the desire for predictability, also drove a subtle difference between Chrome and other browsers of the time: the omnibox dropdown in Chrome contains the complete set of actions we allow the user to take for the current input, and always has a selection, whereas other browsers at the time of Chrome's initial launch treated the dropdown as optional alternates to the address bar input, with no default selection. In a world without inline autocomplete, this makes sense, but with it, such a design would either not show the inline-autocompleted suggestion in the dropdown, or would select the first row of the dropdown only in cases with an inline autocompletion. Both of these lead to irritating behavior inconsistencies (e.g. sometimes having to press different numbers of keystrokes to select the same dropdown row).

Though it's a critically important feature, inline autocompletion has several drawbacks:

- It can distract the user and interfere with typing, especially when autocompleting a low-quality match.
- It is less useful and more frustrating with searches, since searches are shorter (and easier to type) and more variable (so past behavior poorly predicts current behavior).

User complaints around bad autocompletions have almost entirely focused on autocompleted searches.
- It can interact poorly with IMEs and alternate phone keyboards.
- It is limited to cases where the user types a prefix of the desired string. It's difficult to inline autocomplete a navigation to "http://nytimes.com/section/article_title?page=1" when the user input is "nyt article_t". This in turn means it's difficult to make such a navigation the default action for this input, since there's no easy way to indicate this action inline. Finding UX that could remove these limitations, while still allowing users to e.g. delete undesirable "autocompletions" with a single keystroke, would allow aggressive ranking changes.

Users have repeatedly filed bugs asking for a way to turn off inline autocompletion. In principle, it would not be difficult to hack the code to implement this. From a design perspective, however, the centrality of inline autocompletion to the whole omnibox interaction model means that doing this well would require careful study of how all behaviors would be affected and how ranking would have to change. Generally, we've urged users who ask for this to instead clarify the cases where the existing system works poorly, so we can try to improve ranking and behavior for them and all similar users.

Omnibox Modes

Stateful UI complicates user mental models, which is anathema to "reduce cognitive load", so the omnibox tries to avoid having multiple operating modes.

This impacts a number of past and present proposals, including the "origin chip", other security ideas that center around having different "steady state" and "editing" behaviors, and a brainstorming idea of a "fancy state" for things like switching to other tabs. All of these introduce statefulness in order to simplify something else; for example, the origin chip tries to simplify "find the domain I'm on", which users use as a proxy for "is this site trustworthy".

Such tradeoffs result in users paying a continual cognitive cost (to know what mode they're in for any use of the control) in order to reduce some specific costs; therefore, they're only worth making when the specific costs are high enough, or frequent enough, to outweigh the continual cost, and there is no better alternative. To return to the origin chip example, an alternative might be to simply make more aggressive trust decisions within the browser and not require users to evaluate site trustworthiness so often.

That said, the omnibox has subtle statefulness even today. Once users begin editing the omnibox, the code sets a bit called "input in progress"; at this point, underlying navigations will not change the visible text (so as to avoid interrupting editing), and hitting <esc> can be used to bail out and restore the original URL (referred to in the code as the "permanent text"). Arrowing through the dropdown sets another bit, "has temporary text"; now further dropdown updates are frozen, and <esc> resets from "has temporary text" back to the simple "input in progress" state

with the edited text in the omnibox and the first dropdown result selected. (Hitting <esc> again will reset to the permanent text. Hitting <esc> *again* will stop any underlying page load, because that's a global browser accelerator when the omnibox isn't otherwise using it. So it's possible to hit "<esc><esc><esc>" in the omnibox and have each keypress do something different.)

Similarly, triggering tab-to-search sets the "keyword selected" bit, modifying the textfield display and suggestion ranking model, and <esc> can be used to bail out.

This sort of statefulness can be dangerous if users are consciously aware it exists or have to think about how to use it. It's mainly present because users need ways to cancel their current action, and other browsers do similar things; even the choice of <tab> to trigger keyword search mode was meant to imitate tabbing between separate address and search bars (and is why hitting <shift>-<tab> immediately after <tab> will bail out of keyword mode just like <esc>). But it's extremely complicated under the hood; just in writing this section I found and filed two separate bugs, and in general these modes have been the source of much complexity and effort.

Dropdown Styling

Since the omnibox is aimed to maximize speed and minimize cognitive load, the dropdown contents are styled as simply as possible to avoid catching the eye during typing. Because the omnibox should do the right thing by default, and inline autocompletion should make clear what that "right thing" is, users should never need to look down. (This is clearly aspirational rather than completely realistic, or we wouldn't have a dropdown at all.)

Accordingly, the dropdown avoids favicons; minimizes the number of separate icons that *are* used; uses a single font face and size, very few colors, and only "bold" and "normal" styles; gives only a single row of text to each suggestion; and always puts URLs on the same side (the left), so they don't shift around and attract attention. Even the number of results in the dropdown is minimal to make it feel visually light and minimize scanning/parsing time.

The biggest exception to these rules is Answers in Suggest (AiS), which can include custom icons, different font sizes, colors, and styles, and multiple rows. AiS results, when they trigger, can be extremely eye-catching. Effectively, they stop the user in their tracks and demand immediate attention. Thus we have to be careful that they only trigger when that's the effect we really want, and not just any time they might be potentially helpful, lest we undermine the core design goals.

The downside of these optimizations is that the omnibox works poorly for "exploratory" input, where users are unsure exactly which result they want, and are digging through a large number of possibly-similar options. This is most common for navigations to infrequently-visited URLs and URLs that are similar to others (e.g. bugs in a bug tracker, Google docs, leaf pages within

the same site). Perhaps we can detect these cases and transition to a more effective dropdown UI, or create a presentation that serve both kinds of cases simultaneously.

Server-side Ranking

Initial omnibox ranking was almost entirely client-side. Server-side ranking input was limited to ordering the search suggestion results, which the omnibox would decide how to score and mix into local results.

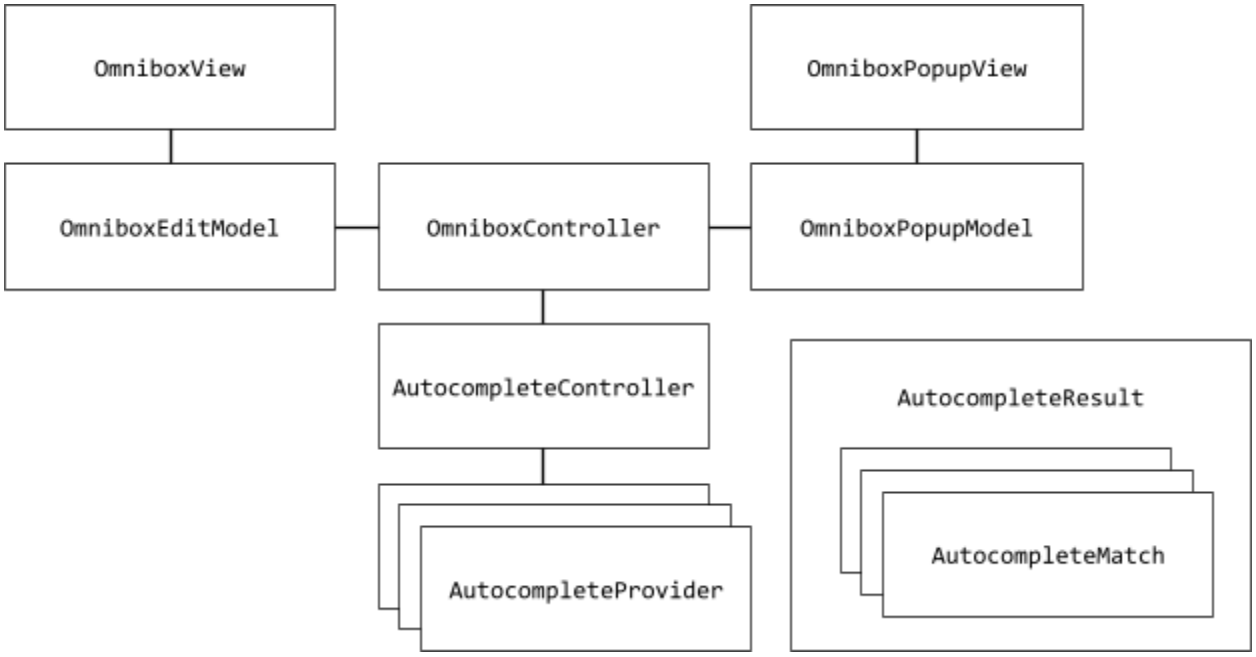
Since search providers like Google can aggregate enormous quantities of data and shard ranking algorithms across whole data centers, it's reasonable to ask if more ranking should happen on the server, and several years ago Google began providing fine-grained relevance scores for search and navigational suggestions. But this approach can make the goals stated earlier harder to achieve:

- **Predictability.** Latency makes it hard for server-side ranking to achieve goals like "no race conditions" and "don't shift dropdown results unnecessarily", especially in markets with poor connectivity.
- **Relevance.** The server generally does not have all the information the client has (e.g. for privacy reasons). Therefore, its suggestions will tend to be less personalized, and thus potentially less accurate.
- **Stability.** To uphold principles like "stability within interactions", servers may need to model the user's current interaction (rather than respond statelessly). If the server fails to do this it can destabilize the behavior. If the server does do this, there is no guarantee its model of the user's interaction matches the omnibox' local conception of user interaction, so stability problems are still possible.
- **Bias.** The goal of the omnibox is to get users where they want to go, not to increase Google searches; and we assume what users frequently want is to navigate. But server-side code at Google is generally maintained by groups which have an inherent interest in users doing more searches, and user benefits are evaluated in terms of effects on searching.

Even with the limited amount of server-side ranking done today, the omnibox code makes significant effort to compensate for the issues above (where possible) to try and preserve its design principles.

Code Architecture Summary

The following is a simplified, idealized version of the core omnibox architecture. Documentation parts 2 and 3 go into far more detail.



The `OmniboxEditModel` and `OmniboxView` classes are responsible for user interaction with the omnibox textfield, while the `OmniboxPopupModel` and `OmniboxPopupView` classes are responsible for the dropdown. These are covered in part 2.

The `AutocompleteController` handles requests to start/stop autocompletion for a particular input sequence. It manages a set of `AutocompleteProviders`, each of which provides zero or more matches for the current input as `AutocompleteMatches`. The `AutocompleteController` then collates these into a single `AutocompleteResult` set. These are covered in part 3.

The `OmniboxController` ties all these together plumbing-wise, communicating initial autocompletion requests from the `OmniboxEditModel` to the `AutocompleteController`, and keeping the edit and popup updated when the current `AutocompleteResult` is updated. This is covered, briefly, in part 2.

Brief History

TODO detail, correct time ordering

- Usage of Firefox flags, customizations to combine boxes
- mconnor “inline autocomplete will never be on by default in any browser”
- Proposal in early Chrome brainstorming
- “Psychic omnibox” name (brakowski)
- Early UI mocks
- Tab-to-search invention (brakowski/glen/acw)
- Implementation (pkasting, sky, brettw)
- Search and navsuggestions (kochi?)

- HistoryContentsProvider
- Introduction of accidental search infobar (brakowski idea)
- Cross-platform porting begins
- UX redesign by alcor
- HistoryQuickProvider (mrossetti)
- ShortcutsProvider (georgey)
- views::Textfield and the elimination of OmniboxViewWin
- mrossetti leaves, mpearson joins?
- Tabbing through dropdown (external contributor)
- HCP removed
- ZeroSuggest (hfung)
- 1993 (TODO link to summary doc?)
- Omnitheatre (TODO link to takeaways doc if it exists?)
- Refactoring (beaudoin)
- AiS (dschuyler, jdonnelly)
- Server-side suggest ranking
- Chrome 51 (May 25, 2016): Material Design redesign of omnibox (and rest of top chrome) first ships on ChromeOS, bringing programmatic rendering of borders/icons/etc.
- Chrome 52 (July 20, 2016): New security icons first ship on Mac as part of the Mac "top chrome MD" rollout: https://docs.google.com/document/u/2/d/1jIfCjcsZUL6ouLgPOsMORGcTTXc7OTwkcBQCkZvDyGE/pub .
- Chrome 55 (Dec. 1, 2016): Verbose security states: "Secure" on HTTPS, "Not secure" or "Dangerous" in particular cases. This is a precursor to "HTTP Bad", the effort to explicitly mark HTTP sites as "not secure" (at which point the "secure" marking on HTTPS can probably be dropped).
- Dec. 6, 2016: Full-time omnibox team formally created.
- Chrome 56 (Jan. 25, 2017): "HTTP Bad" phase 1: HTTP sites explicitly marked "not secure" when containing password or credit card fields.

Other Documentation

Documents in this section are outdated, but may still be of interest.

The State of the Omnibox: March 2012
During the 1993 project, PM Gideon Wald attempted to assemble an initial "omnibox design doc". While this was never completed, it serves as a good "overview" document, and provided valuable reminders of key topics to cover in this documentation.

Simplified and Detailed Omnibox Code Flow
These drawings cover what happens when users press a key in the omnibox.

Post-1993 launch omnibox and refactor

After 1993 launched, Philippe Beaudoin began trying to refactor the omnibox code to improve a variety of problems with the design and implementation. This doc succinctly describes the current system state, enumerates its problems, and lays out a plan to improve it. Some of the steps of this plan have been subsequently implemented, but many remain to be done. Anyone interested in maintaining the omnibox should start here.