- Text Fragments
  - Browser compatibility
  - start
  - start and end
  - prefix- and -suffix
  - The full syntax
  - Creating Text Fragment URLs with a browser extension
  - Multiple text fragments in one URL
  - Mixing element and text fragments
  - The fragment directive
  - Feature detection
  - Styling text fragments
  - Polyfillability
  - Programmatic Text Fragment link generation
  - Obtaining Text Fragments for analytics purposes
  - Security
  - Privacy
- Disabling text fragments
- Text fragments in web search
- Conclusion
- Related links
- Acknowledgements

Chrome 80 was a big release. It contained a number of highly anticipated features like ECMAScript Modules in Web Workers, nullish coalescing, optional chaining, and more. The release was, as usual, announced through a blog post on the Chromium blog. You can see an excerpt of the blog post in the screenshot below.

{% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/egsW6tkKWYI8IHE6JyMZ.png", alt="", width="400", height="628" %}
Chromium blog post with red boxes around elements with an `id` attribute.

You are probably asking yourself what all the red boxes mean. They are the result of running the following snippet in DevTools. It highlights all elements that have an `id` attribute.

```
document.querySelectorAll('[id]').forEach((el) => {
  el.style.border = 'solid 2px red';
});
```

I can place a deep link to any element highlighted with a red box thanks to the fragment identifier which I then use in the hash of the page's URL. Assuming I wanted to deep link to the *Give us feedback in our Product Forums* box in the aside, I could do so by handcrafting the URL `https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#HTML1`. As you can see in the Elements panel of the Developer Tools, the element in question has an `id` attribute with the value `HTML1`.

{% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/whVXhhrYwA55S3i4J3l5.png", alt="", width="600", height="97" %}
Dev Tools showing the `id` of an element.

If I parse this URL with JavaScript's `URL()` constructor, the different components are revealed. Notice the `hash` property with the value `#HTML1`.

```
new URL('https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#HTML1');
/* Creates a new `URL` object
URL {
  hash: "#HTML1"
  host: "blog.chromium.org"
  hostname: "blog.chromium.org"
  href: "https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#HTML1"
  origin: "https://blog.chromium.org"
  password: ""
  pathname: "/2019/12/chrome-80-content-indexing-es-modules.html"
  port: ""
  protocol: "https:"
  search: ""
  searchParams: URLSearchParams {}
  username: ""
}
*/
```

The fact though that I had to open the Developer Tools to find the `id` of an element speaks volumes about the probability this particular section of the page was meant to be linked to by the author of the blog post.

What if I want to link to something without an `id`? Say I want to link to the *ECMAScript Modules in Web Workers* heading. As you can see in the screenshot below, the `<h1>` in question does not have an `id` attribute, meaning there is no way I can link to this heading. This is the problem that Text Fragments solve.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/1g4rTS1q5LKHEHnDoF9o.png", alt="", width="600", height="71" %}
> Dev Tools showing a heading without an `id`.

Text Fragments

The Text Fragments proposal adds support for specifying a text snippet in the URL hash. When navigating to a URL with such a text fragment, the user agent can emphasize and/or bring it to the user's attention.

Browser compatibility

{% BrowserCompat 'css.selectors.target-text' %}

{% Aside 'success' %} These links used to not work when served across client-side redirects that some common services like Twitter use. This issue was tracked as crbug.com/1055455 and is now fixed. Regular HTTP redirects always worked fine. {% endAside %}

For security reasons, the feature requires links to be opened in a `noopener` context. Therefore, make sure to include `rel="noopener"` in your `<a>` anchor markup or add `noopener` to your `Window.open()` list of window functionality features.

`start`

In its simplest form, the syntax of Text Fragments is as follows: The hash symbol `#` followed by `:~:text=` and finally `start`, which represents the percent-encoded text I want to link to.

```
#:~:text=start
```

For example, say that I want to link to the *ECMAScript Modules in Web Workers* heading in the blog post announcing features in Chrome 80, the URL in this case would be:

`https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#:~:text=ECMAScript%20Modules%20in%20Web%20Workers`

The text fragment is emphasized like this. If you click the link in a supporting browser like Chrome, the text fragment is highlighted and scrolls into view:

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/D3jwPrJlvN3FmJo3pADt.png", alt="", width="400", height="208" %}
> Text fragment scrolled into view and highlighted.

`start` and `end`

Now what if I want to link to the entire *section* titled *ECMAScript Modules in Web Workers*, not just its heading? Percent-encoding the entire text of the section would make the resulting URL impracticably long.

Luckily there is a better way. Rather than the entire text, I can frame the desired text using the `start,end` syntax. Therefore, I specify a couple of percent-encoded words at the beginning of the desired text, and a couple of percent-encoded words at the end of the desired text, separated by a comma `,`.

That looks like this:

`https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#:~:text=ECMAScript%20Modules%20in%20Web%20Workers,ES%20Modules%20in%20Web%20Workers.`.

For `start`, I have `ECMAScript%20Modules%20in%20Web%20Workers`, then a comma `,` followed by `ES%20Modules%20in%20Web%20Workers.` as `end`. When you click through on a supporting browser like Chrome, the whole section is highlighted and scrolled into view:

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/2yTYmKnjHTnqXkcmHF1F.png", alt="", width="400", height="343" %}
> Text fragment scrolled into view and highlighted.

Now you may wonder about my choice of `start` and `end`. Actually, the slightly shorter URL `https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#:~:text=ECMAScript%20Modules,Web%20Workers.` with only two words on each side would have worked, too. Compare `start` and `end` with the previous values.

If I take it one step further and now use only one word for both `start` and `end`, you can see that I am in trouble. The URL `https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#:~:text=ECMAScript,Workers.` is even shorter now, but the highlighted text fragment is no longer the originally desired one. The highlighting stops at the first occurrence of the word

`Workers.`, which is correct, but not what I intended to highlight. The problem is that the desired section is not uniquely identified by the current one-word `start` and `end` values:

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/GGbbtHBpsoFyubnISyZw.png", alt="", width="400", height="342" %}
> Non-intended text fragment scrolled into view and highlighted.

`prefix-` and `-suffix`

Using long enough values for `start` and `end` is one solution for obtaining a unique link. In some situations, however, this is not possible. On a side note, why did I choose the Chrome 80 release blog post as my example? The answer is that in this release Text Fragments were introduced:

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/yA1p3CijeDbTRwMys9Hq.png", alt="Blog post text: Text URL Fragments. Users or authors can now link to a specific portion of a page using a text fragment provided in a URL. When the page is loaded, the browser highlights the text and scrolls the fragment into view. For example, the URL below loads a wiki page for 'Cat' and scrolls to the content listed in the `text` parameter.", width="800", height="200" %}
> Text Fragments announcement blog post excerpt.

Notice how in the screenshot above the word "text" appears four times. The forth occurrence is written in a green code font. If I wanted to link to this particular word, I would set `start` to `text`. Since the word "text" is, well, only one word, there cannot be a `end`. What now? The URL `https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#:~:text=text` matches at the first occurrence of the word "Text" already in the heading:

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/nXxCskUwdCxwxejPSSZW.png", alt="", width="800", height="209" %}
> Text Fragment matching at the first occurrence of "Text".

{% Aside 'caution' %} Note that text fragment matching is case-insensitive. {% endAside %}

Luckily there is a solution. In cases like this, I can specify a `prefix-` and a `-suffix`. The word before the green code font "text" is "the", and the word after is "parameter". None of the other three occurrences of the word "text" has the same surrounding words. Armed with this knowledge, I can tweak the previous URL and add the `prefix-` and the `-suffix`. Like the other parameters, they, too, need to be percent-encoded and can contain more than one word. `https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#:~:text=the-,text,-parameter`. To allow the parser to clearly identify the `prefix-` and the `-suffix`, they need to be separated from the `start` and the optional `end` with a dash `-`.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/J3L5BVSMmzGY6xdkabP6.png", alt="", width="800", height="203" %}
> Text Fragment matching at the desired occurrence of "text".

The full syntax

The full syntax of Text Fragments is shown below. (Square brackets indicate an optional parameter.) The values for all parameters need to be percent-encoded. This is especially important for the dash `-`, ampersand `&`, and comma `,` characters, so they are not being interpreted as part of the text directive syntax.

```
#:~:text=[prefix-,]start[,end][,-suffix]
```

Each of `prefix-`, `start`, `end`, and `-suffix` will only match text within a single block-level element, but full `start,end` ranges *can* span multiple blocks. For example, `:~:text=The quick,lazy dog` will fail to match in the following example, because the starting string "The quick" does not appear within a single, uninterrupted block-level element:

```
<div>
  The
  <div></div>
  quick brown fox
</div>
<div>jumped over the lazy dog</div>
```

It does, however, match in this example:

```
<div>The quick brown fox</div>
<div>jumped over the lazy dog</div>
```

Creating Text Fragment URLs with a browser extension

Creating Text Fragments URLs by hand is tedious, especially when it comes to making sure they are unique. If you really want to, the specification has some tips and lists the exact steps for generating Text Fragment URLs. We provide an open-source browser extension called Link to Text Fragment that lets you link to any text by selecting it, and then clicking "Copy Link to Selected Text" in the context menu. This extension is available for the following browsers:

- Link to Text Fragment for Google Chrome
- Link to Text Fragment for Microsoft Edge
- Link to Text Fragment for Mozilla Firefox

- Link to Text Fragment for Apple Safari

{% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/ASLtFCPoHvyTKrAtKAv4.png", alt="", width="800", height="500" %}

Link to Text Fragment browser extension.

Multiple text fragments in one URL

Note that multiple text fragments can appear in one URL. The particular text fragments need to be separated by an ampersand character `&`. Here is an example link with three text fragments: `https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#:~:text=Text%20URL%20Fragments&text=text,-parameter&text=:~:text=On%20islands,%20birds%20can%20contribute%20as%20much%20as%2060%25%20of%20a%20cat's%20diet`.

{% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/ffsq7aoSoVd9q6r5cquY.png", alt="", width="800", height="324" %}
Three text fragments in one URL.

Mixing element and text fragments

Traditional element fragments can be combined with text fragments. It is perfectly fine to have both in the same URL, for example, to provide a meaningful fallback in case the original text on the page changes, so that the text fragment does not match anymore. The URL `https://blog.chromium.org/2019/12/chrome-80-content-indexing-es-modules.html#HTML1:~:text=Give%20us%20feedback%20in%20our%20Product%20Forums.` linking to the *Give us feedback in our Product Forums* section contains both an element fragment (`HTML1`), as well as a text fragment (`text=Give%20us%20feedback%20in%20our%20Product%20Forums.`):

{% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/JRKCM6Ihrq8sgRZRiymr.png", alt="", width="237", height="121" %}
Linking with both element fragment and text fragment.

The fragment directive

There is one element of the syntax I have not explained yet: the fragment directive `:~:`. To avoid compatibility issues with existing URL element fragments as shown above, the Text Fragments specification introduces the fragment directive. The fragment directive is a portion of the URL fragment delimited by the code sequence `:~:`. It is reserved for user agent instructions, such as `text=`, and is stripped from the URL during loading so that author scripts cannot directly interact with it. User agent instructions are also called *directives*. In the concrete case, `text=` is therefore called a *text directive*.

Feature detection

To detect support, test for the read-only `fragmentDirective` property on `document`. The fragment directive is a mechanism for URLs to specify instructions directed to the browser rather than the document. It is meant to avoid direct interaction with author script, so that future user agent instructions can be added without fear of introducing breaking changes to existing content. One potential example of such future additions could be translation hints.

```
if ('fragmentDirective' in document) {
  // Text Fragments is supported.
}
```

{% Aside %} From Chrome 80 to Chrome 85, the `fragmentDirective` property was defined on `Location.prototype`. For details on this change, see WICG/scroll-to-text-fragment#130. {% endAside %}

Feature detection is mainly intended for cases where links are dynamically generated (for example by search engines) to avoid serving text fragments links to browsers that do not support them.

Styling text fragments

By default, browsers style text fragments the same way they style `mark` (typically black on yellow, the CSS system colors for `mark`). The user-agent stylesheet contains CSS that looks like this:

```
:root::target-text {
  color: MarkText;
  background: Mark;
}
```

As you can see, the browser exposes a pseudo selector `::target-text` that you can use to customize the applied highlighting. For example, you could design your text fragments to be black text on a red background. As always, be sure to check the color contrast so your override styling does not cause accessibility issues and make sure the highlighting actually visually stands out from the rest of the content.

```
:root::target-text {
  color: black;
  background-color: red;
}
```

Polyfillability

The Text Fragments feature can be polyfilled to some extent. We provide a polyfill, which is used internally by the extension, for browsers that do not provide built-in support for Text Fragments where the functionality is implemented in JavaScript.

Programmatic Text Fragment link generation

The polyfill contains a file `fragment-generation-utils.js` that you can import and use to generate Text Fragment links. This is outlined in the code sample below:

```
const { generateFragment } = await import('https://unpkg.com/text-fragments-
polyfill/dist/fragment-generation-utils.js');
const result = generateFragment(window.getSelection());
if (result.status === 0) {
  let url = `${location.origin}${location.pathname}${location.search}`;
  const fragment = result.fragment;
  const prefix = fragment.prefix ?
    `${encodeURIComponent(fragment.prefix)}-,` :
    '';
  const suffix = fragment.suffix ?
    `,-${encodeURIComponent(fragment.suffix)}` :
    '';
  const start = encodeURIComponent(fragment.textStart);
  const end = fragment.textEnd ?
    `,${encodeURIComponent(fragment.textEnd)}` :
    '';
  url += `#:~:text=${prefix}${start}${end}${suffix}`;
  console.log(url);
}
```

Obtaining Text Fragments for analytics purposes

Plenty of sites use the fragment for routing, which is why browsers strip out Text Fragments so as to not break those pages. There is an acknowledged need to expose Text Fragments links to pages, for example, for analytics purposes, but the proposed solution is not implemented yet. As a workaround for now, you can use the code below to extract the desired information.

```
new URL(performance.getEntries().find(({ type }) => type === 'navigate').name).hash;
```

Security

Text fragment directives are invoked only on full (non-same-page) navigations that are the result of a user activation. Additionally, navigations originating from a different origin than the destination will require the navigation to take place in a `noopener` context, such that the destination page is known to be sufficiently isolated. Text fragment directives are only applied to the main frame. This means that text will not be searched inside iframes, and iframe navigation will not invoke a text fragment.

Privacy

It is important that implementations of the Text Fragments specification do not leak whether a text fragment was found on a page or not. While element fragments are fully under the control of the original page author, text fragments can be created by anyone. Remember how in my example above there was no way to link to the *ECMAScript Modules in Web Workers* heading, since the `<h1>` did not have an `id`, but how anyone, including me, could just link to anywhere by carefully crafting the text fragment?

Imagine I ran an evil ad network `evil-ads.example.com`. Further imagine that in one of my ad iframes I dynamically created a hidden cross-origin iframe to `dating.example.com` with a Text Fragment URL `dating.example.com#:~:text=Log%20Out` once the user interacts with the ad. If the text "Log Out" is found, I know the victim is currently logged in to `dating.example.com`, which I could use for user profiling. Since a naive Text Fragments implementation might decide that a successful match should cause a focus switch, on `evil-ads.example.com` I could listen for the `blur` event and thus know when a match occurred. In Chrome, we have implemented Text Fragments in such a way that the above scenario cannot happen.

Another attack might be to exploit network traffic based on scroll position. Assume I had access to network traffic logs of my victim, like as the admin of a company intranet. Now imagine there existed a long human resources document *What to Do If You Suffer From...* and then a list of conditions like *burn out*, *anxiety*, etc. I could place a tracking pixel next to each item on the list. If I then determine that loading the document temporally co-occurs with the loading of the tracking pixel next to, say, the *burn out* item, I can then, as the intranet admin, determine that an employee has clicked through on a text fragment link with `:~:text=burn%20out` that the employee may have assumed was confidential and not visible to anyone. Since this example is somewhat contrived to

begin with and since its exploitation requires *very* specific preconditions to be met, the Chrome security team evaluated the risk of implementing scroll on navigation to be manageable. Other user agents may decide to show a manual scroll UI element instead.

For sites that wish to opt-out, Chromium supports a Document Policy header value that they can send so user agents will not process Text Fragment URLs.

```
Document-Policy: force-load-at-top
```

Disabling text fragments

The easiest way for disabling the feature is by using an extension that can inject HTTP response headers, for example, ModHeader (not a Google product), to insert a response (*not* request) header as follows:

```
Document-Policy: force-load-at-top
```

Another, more involved, way to opt out is by using the enterprise setting `ScrollToTextFragmentEnabled`. To do this on macOS, paste the command below in the terminal.

```
defaults write com.google.Chrome ScrollToTextFragmentEnabled -bool false
```

On Windows, follow the documentation on the Google Chrome Enterprise Help support site.

{% Aside 'warning' %} Only try this when you know what you are doing. {% endAside %}

Text fragments in web search

For some searches, the search engine Google provides a quick answer or summary with a content snippet from a relevant website. These *featured snippets* are most likely to show up when a search is in the form of a question. Clicking a featured snippet takes the user directly to the featured snippet text on the source web page. This works thanks to automatically created Text Fragments URLs.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/KbZgnGxZOOymLxYPZyGH.png", alt="", width="800", height="451" %}
> Google search engine results page showing a featured snippet. The status bar shows the Text Fragments URL.

{% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/4Q7zk9xBnb2uw8GRaLnU.png", alt="", width="800", height="451" %}
After clicking through, the relevant section of the page is scrolled into view.

Conclusion

Text Fragments URL is a powerful feature to link to arbitrary text on webpages. The scholarly community can use it to provide highly accurate citation or reference links. Search engines can use it to deeplink to text results on pages. Social networking sites can use it to let users share specific passages of a webpage rather than inaccessible screenshots. I hope you start using Text Fragment URLs and find them as useful as I do. Be sure to install the Link to Text Fragment browser extension.

Related links

- Spec draft
- TAG Review
- Chrome Platform Status entry
- Chrome tracking bug
- Intent to Ship thread
- WebKit-Dev thread
- Mozilla standards position thread

Acknowledgements

Text Fragments was implemented and specified by Nick Burris and David Bokan, with contributions from Grant Wang. Thanks to Joe Medley for the thorough review of this article. Hero image by Greg Rakozy on Unsplash.

□□□ #2
□ □□□ □□□ 2026-10-31 22:21:10 UTC
□ □□□ □□□ 2026-10-31 22:21:12 UTC