

Window: popstate event



Baseline Widely available



The **popstate** event of the **Window** interface is fired when the active history entry changes while the user navigates the session history. It changes the current history entry to that of the last page the user visited or, if `history.pushState()` has been used to add a history entry to the history stack, that history entry is used instead.

Syntax

Use the event name in methods like `addEventListener()`, or set an event handler property.

JS

```
addEventListener("popstate", (event) => { })
```

```
onpopstate = (event) => { }
```

Event type

A **PopStateEvent**. Inherits from **Event**.



Event properties

PopStateEvent.state Read only

Returns a copy of the information that was provided to `pushState()` or `replaceState()`.

Event handler aliases

In addition to the **Window** interface, the event handler property **onpopstate** is also available on the following elements:

- HTMLBodyElement**
- HTMLFrameSetElement**

- `SVGSVGElement`

The history stack

If the history entry being activated was created by a call to `history.pushState()` or was affected by a call to `history.replaceState()`, the `popstate` event's `state` property contains a copy of the history entry's state object.

These methods and their corresponding events can be used to add data to the history stack which can be used to reconstruct a dynamically generated page, or to otherwise alter the state of the content being presented while remaining on the same `Document`.

Note that just calling `history.pushState()` or `history.replaceState()` won't trigger a `popstate` event. The `popstate` event will be triggered by doing a browser action such as a click on the back or forward button (or calling `history.back()` or `history.forward()` in JavaScript).

🕒 **Note:** When writing functions that process `popstate` event it is important to take into account that properties like `window.location` will already reflect the state change (if it affected the current URL), but `document` might still not. If the goal is to catch the moment when the new document state is already fully in place, a zero-delay `setTimeout()` method call should be used to effectively put its inner *callback* function that does the processing at the end of the browser event loop: `window.onpopstate = () => setTimeout(doSomething, 0);`

When popstate is sent

It's important to first understand that — to combat unwanted pop-ups — browsers may not fire the `popstate` event at all unless the page has been interacted with.

This section describes the steps that browsers follow in the cases where they *do* potentially fire the `popstate` event (that is, in the cases where the page has been interacted with).

When a navigation occurs — either due to the user triggering the browser's `Back` button or otherwise — the `popstate` event is near the end of the process to navigate to the new location. It happens after the new location has loaded (if needed), displayed, made visible, and so on — after the `pageshow` event is sent, but before the persisted user state information is restored and the `hashchange` event is sent.

To better understand when the `popstate` event is fired, consider this simplified sequence of events that occurs when the current history entry changes due to either the user navigating the site or the history being traversed programmatically. Here, the transition is changing the current history entry to one we'll refer to as **new-entry**. The current page's session history stack entry will be referred to as **current-entry**.

1. If **new-entry** doesn't currently contain an existing `Document`, fetch the content and create its `Document` before continuing. This will eventually send events such as `DOMContentLoaded` and `load` to the `Window` containing the document, but the steps below will continue to execute in the meantime.
2. If **current-entry**'s title wasn't set using one of the History API methods (`pushState()` or `replaceState()`), set the entry's title to the string returned by its `document.title` attribute.
3. If the browser has state information it wishes to store with the **current-entry** before navigating away from it, it then does so. The entry is now said to have "persisted user state." This information the browser might add to the history session entry may include, for instance, the document's scroll position, the values of form inputs, and other such data.

4. If **new-entry** has a different `Document` object than **current-entry**, the browsing context is updated so that its `document` property refers to the document referred to by **new-entry**, and the context's name is updated to match the context name of the now-current document.
5. Each form control within **new-entry**'s `Document` that has `autocomplete` configured with its autofill field name set to `off` is reset. See [The HTML autocomplete attribute](#) for more about the autocomplete field names and how autocomplete works.
6. If **new-entry**'s document is already fully loaded and ready—that is, its `readyState` is `complete`—and the document is not already visible, it's made visible and the `pageshow` event is fired at the document with the `PageTransitionEvent`'s `persisted` attribute set to `true`.
7. The document's `URL` is set to that of **new-entry**.
8. If the history traversal is being performed with replacement enabled, the entry immediately prior to the destination entry (taking into account the `delta` parameter on methods such as `go()`) is removed from the history stack.
9. If the **new-entry** doesn't have persisted user state and its URL's fragment is non-`null`, the document is scrolled to that fragment.
10. Next, **current-entry** is set to **new-entry**. The destination entry is now considered to be current.
11. If **new-entry** has serialized state information saved with it, that information is deserialized into `History.state`; otherwise, `state` is `null`.
12. If the value of `state` changed, the `popstate` event is sent to the document.
13. Any persisted user state is restored, if the browser chooses to do so.
14. If the original and new entries shared the same document, but had different fragments in their URLs, send the `hashchange` event to the window.

As you can see, the `popstate` event is nearly the last thing done in the process of navigating pages in this way.

Examples

A page at `http://example.com/example.html` running the following code will generate logs as indicated:

JS

```
window.addEventListener("popstate", (event) => {
  console.log(
    `location: ${document.location}, state: ${JSON.stringify(event.state)}`,
  );
});

history.pushState({ page: 1 }, "title 1", "?page=1");
history.pushState({ page: 2 }, "title 2", "?page=2");
history.replaceState({ page: 3 }, "title 3", "?page=3");
history.back(); // Logs "location: http://example.com/example.html?page=1, state: {"page":1}"
history.back(); // Logs "location: http://example.com/example.html, state: null"
history.go(2); // Logs "location: http://example.com/example.html?page=3, state: {"page":3}"
```

The same example using the `onpopstate` event handler property:

JS

```
window.onpopstate = (event) => {
  console.log(
    `location: ${document.location}, state: ${JSON.stringify(event.state)}`,
  );
};

history.pushState({ page: 1 }, "title 1", "?page=1");
history.pushState({ page: 2 }, "title 2", "?page=2");
history.replaceState({ page: 3 }, "title 3", "?page=3");
history.back(); // Logs "location: http://example.com/example.html?page=1, state: {"page":1}"
history.back(); // Logs "location: http://example.com/example.html, state: null"
history.go(2); // Logs "location: http://example.com/example.html?page=3, state: {"page":3}"
```

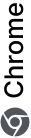
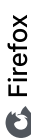
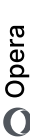
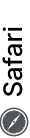
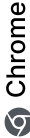
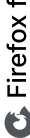
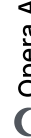
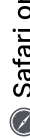
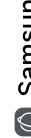
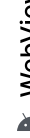
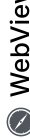
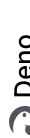
Note that even though the original history entry (for `http://example.com/example.html`) has no state object associated with it, a `popstate` event is still fired when we activate that entry after the second call to `history.back()`.

Specifications

Specification
HTML # event-popstate
HTML # handler-window-onpopstate

Browser compatibility

[Report problems with this compatibility data](#) • [View data on GitHub](#)

													
													
<code>popstate</code> event	✓ 5 *	✓ 12	✓ 4	✓ 11.5	✓ 5 *	✓ 18 *	✓ 4	✓ 11.5	✓ 4.2 *	✓ 1 *	✓ 4.4 *	✓ 4.2 *	✗ No

✓ Full support ✗ No support * See implementation notes.

See also

- [Manipulating the browser history \(the History API\)](#)
- Window: `hashchange` event



Your blueprint for a better internet.