Window: postMessage() method

- Syntax
  - Parameters
  - Return value
- The dispatched event
- Security concerns
  - Secure shared memory messaging
- Examples
  - Notes
  - Using window.postMessage in extensions
- Specifications
- Browser compatibility
- See also

The **`window.postMessage()`** method safely enables cross-origin communication between Window objects; *e.g.,* between a page and a pop-up that it spawned, or between a page and an iframe embedded within it.

Normally, scripts on different pages are allowed to access each other if and only if the pages they originate from share the same origin (also known as the "same-origin policy"). `window.postMessage()` provides a controlled mechanism to securely circumvent this restriction (if used properly).

Furthermore, an accessing script must have obtained the window object of the accessed document beforehand. This can occur through methods such as `window.open()` for popups or `iframe.contentWindow` for iframes.

Broadly, one window may obtain a reference to another (*e.g.,* via `targetWindow = window.opener`), and then dispatch a MessageEvent on it with `targetWindow.postMessage()`. The receiving window is

then free to handle this event as needed. The arguments passed to `window.postMessage()` (*i.e.*, the "message") are exposed to the receiving window through the event object.

Syntax

```
postMessage(message)
postMessage(message, targetOrigin)
postMessage(message, targetOrigin, transfer)

postMessage(message, options)
```

Parameters

- `message`
  - : Data to be dispatched to the other window. The data is serialized using the structured clone algorithm. This means you can pass a broad variety of data objects safely to the destination window without having to serialize them yourself.
- `targetOrigin`
  - : Specifies the origin the recipient window must have in order to receive the event. In order for the event to be dispatched, the origin must match exactly (including scheme, hostname, and port). If omitted, it defaults to `"/"`, which is the origin that is calling the method. This mechanism provides control over where messages are sent; for example, if `postMessage()` was used to transmit a password, it would be absolutely critical that this argument be a URI whose origin is the same as the intended receiver of the message containing the password, to prevent interception of the password by a malicious third party. `*` may also be provided, which means the message can be dispatched to a listener with any origin.

    > **NOTE:** Always provide a specific `targetOrigin`, not `*`, if you know where the other window's document should be located. Failing to provide a specific target could disclose data to a malicious site. Because `data:` URLs have opaque origins, in order to send messages to a context with a `data:` URL, you must specify `"*"`.

- `transfer`
  - : An optional array of transferable objects to transfer ownership of. The ownership of these objects is given to the destination side and they are no longer usable on the sending side. These transferable objects are not automatically sent; they must either be contained in the message or be accessible to the recipient via other means, such

as MessagePort via MessageEvent.ports.

- `options`
  - : An optional object containing the following properties:
    - `transfer`
      - : Has the same meaning as the `transfer` parameter.
    - `targetOrigin`
      - : Has the same meaning as the `targetOrigin` parameter.

Return value

None (undefined).

The dispatched event

A `window` can listen for dispatched messages by executing the following JavaScript:

```
window.addEventListener("message", (event) => {
  if (event.origin !== "http://example.org:8080") return;

  // …
});
```

The properties of the dispatched message are:

- `data`
  - : The object passed from the other window.
- `origin`
  - : The origin of the window that sent the message at the time `postMessage` was called. This string is the concatenation of the protocol and "://", the host name if one exists, and ":" followed by a port number if a port is present and differs from the default port for the given protocol. Examples of typical origins are `https://example.org` (implying port `443`), `http://example.net` (implying port `80`), and `http://example.com:8080`. Note that this origin is *not* guaranteed to be the current or future origin of that window, which might have been navigated to a different location since `postMessage` was called.
- `source`
  - : A reference to the window object that sent the message; you can use this to establish two-way communication between two windows with different origins.

Security concerns

If you do not expect to receive messages from other sites, *do not* add any event listeners for `message` events. This is a completely foolproof way to avoid security problems.

If you do expect to receive messages from other sites, **always verify the sender's identity** using the `origin` and possibly `source` properties. Any window (including, for example, `http://evil.example.com`) can send a message to any other window within the iframe hierarchy from top to every iframe below of the current document. Having verified identity, however, you still should **always verify the syntax of the received message**. Otherwise, a security hole in the site you trusted to send only trusted messages could then open a cross-site scripting hole in your site.

Always specify an exact target origin, not `*`, when you use `postMessage` to dispatch data to other windows. A malicious site can change the location of the window without your knowledge, and therefore it can intercept the data sent using `postMessage`.

Secure shared memory messaging

If `postMessage()` throws when used with SharedArrayBuffer objects, you might need to make sure you cross-site isolated your site properly. Shared memory is gated behind two HTTP headers:

- Cross-Origin-Opener-Policy with `same-origin` as value (protects your origin from attackers)
- Cross-Origin-Embedder-Policy with `require-corp` or `credentialless` as value (protects victims from your origin)

```
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Embedder-Policy: require-corp
```

To check if cross-origin isolation has been successful, you can test against the Window.crossOriginIsolated property available to window and worker contexts:

```
const myWorker = new Worker("worker.js");

if (crossOriginIsolated) {
  const buffer = new SharedArrayBuffer(16);
  myWorker.postMessage(buffer);
} else {
  const buffer = new ArrayBuffer(16);
```

```
  myWorker.postMessage(buffer);
}
```

Examples

```
/*
 * In window A's scripts, with A being on http://example.com:8080:
 */

const popup = window.open(/* popup details */);

// When the popup has fully loaded, if not blocked by a popup blocker:

// This does nothing, assuming the window hasn't changed its location.
popup.postMessage(
  "The user is 'bob' and the password is 'secret'",
  "https://secure.example.net",
);

// This will successfully queue a message to be dispatched to the popup, assuming
// the window hasn't changed its location.
popup.postMessage("hello there!", "http://example.com");

window.addEventListener("message", (event) => {
  // Do we trust the sender of this message? (might be
  // different from what we originally opened, for example).
  if (event.origin !== "http://example.com") return;

  // event.source is popup
  // event.data is "hi there yourself! the secret response is: rheeeeet!"
});
```

```
/*
 * In the popup's scripts, running on http://example.com:
 */

// Called sometime after postMessage is called
```

```
window.addEventListener("message", (event) => {
  // Do we trust the sender of this message?
  if (event.origin !== "http://example.com:8080") return;

  // event.source is window.opener
  // event.data is "hello there!"

  // Assuming you've verified the origin of the received message (which
  // you must do in any case), a convenient idiom for replying to a
  // message is to call postMessage on event.source and provide
  // event.origin as the targetOrigin.
  event.source.postMessage(
    "hi there yourself! the secret response is: rheeeeet!",
    event.origin,
  );
});
```

Notes

Any script in a document in a window can request to let a message get dispatched to a document in another window whose window object it has obtained, by calling `.postMessage()` on that window object. Consequently, any event listener used to receive messages **must** first check the identity of the sender of the message, using the `origin` and possibly `source` properties. This cannot be overstated: **Failure to check the `origin` and possibly `source` properties enables cross-site scripting attacks.**

As with any asynchronously-dispatched script (timeouts, user-generated events), it is not possible for the caller of `postMessage` to detect when an event handler listening for events sent by `postMessage` throws an exception.

After `postMessage()` is called, the MessageEvent will be dispatched *only after all pending execution contexts have finished*. For example, if `postMessage()` is invoked in an event handler, that event handler will run to completion, as will any remaining handlers for that same event, before the MessageEvent is dispatched.

The value of the `origin` property of the dispatched event is not affected by the current value of `document.domain` in the calling window.

For IDN host names only, the value of the `origin` property is not consistently Unicode or punycode; for greatest compatibility check for both the IDN and punycode values when using this property if you expect messages from IDN sites. This value will eventually be consistently IDN, but for now you should handle both IDN and punycode forms.

The value of the `origin` property when the sending window contains a `javascript:` or `data:` URL is the origin of the script that loaded the URL.

Using window.postMessage in extensions

`window.postMessage` is available to JavaScript running in chrome code (e.g., in extensions and privileged code), but the `source` property of the dispatched event is always `null` as a security restriction. (The other properties have their expected values.)

It is not possible for content or web context scripts to specify a `targetOrigin` to communicate directly with an extension (either the background script or a content script). Web or content scripts *can* use `window.postMessage` with a `targetOrigin` of `"*"` to broadcast to every listener, but this is discouraged, since an extension cannot be certain the origin of such messages, and other listeners (including those you do not control) can listen in.

Content scripts should use runtime.sendMessage to communicate with the background script. Web context scripts can use custom events to communicate with content scripts (with randomly generated event names, if needed, to prevent snooping from the guest page).

Lastly, dispatching a message to a page at a `file:` URL currently requires that the `targetOrigin` argument be `"*"`. `file://` cannot be used as a security restriction; this restriction may be modified in the future.

Specifications

Browser compatibility

See also

- Document.domain
- CustomEvent
- BroadcastChannel - For same-origin communication.

□□□ #3
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ cocobook-sync □□□ 2026-11-04 17:35:16 UTC