`<webview>` Tag

Warning

Electron's `webview` tag is based on Chromium's `webview`, which is undergoing dramatic architectural changes. This impacts the stability of `webviews`, including rendering, navigation, and event routing. We currently recommend to not use the `webview` tag and to consider alternatives, like `iframe`, Electron's `BrowserView`, or an architecture that avoids embedded content altogether.

Enabling

By default the `webview` tag is disabled in Electron >= 5. You need to enable the tag by setting the `webviewTag` webPreferences option when constructing your `BrowserWindow`. For more information see the BrowserWindow constructor docs.

Overview

> *Display external web content in an isolated frame and process.*

Process: Renderer
This class is not exported from the `'electron'` module. It is only available as a return value of other methods in the Electron API.

Use the `webview` tag to embed 'guest' content (such as web pages) in your Electron app. The guest content is contained within the `webview` container. An embedded page within your app controls how the guest content is laid out and rendered.

Unlike an `iframe`, the `webview` runs in a separate process than your app. It doesn't have the same permissions as your web page and all interactions between your app and embedded content will be asynchronous. This keeps your app safe from the embedded content. **Note:** Most methods called on the webview from the host page require a synchronous call to the main process.

Example

To embed a web page in your app, add the `webview` tag to your app's embedder page (this is the app page that will display the guest content). In its simplest form, the `webview` tag includes the `src` of the web page and css styles that control the appearance of the `webview` container:

```
<webview id="foo" src="https://www.github.com/" style="display:inline-flex;
width:640px; height:480px"></webview>
```

If you want to control the guest content in any way, you can write JavaScript that listens for `webview` events and responds to those events using the `webview` methods. Here's sample code with two event listeners: one that listens for the web page to start loading, the other for the web page to stop loading, and displays a "loading..." message during the load time:

```
<script>
  onload = () => {
    const webview = document.querySelector('webview')
```

```
    const indicator = document.querySelector('.indicator')

    const loadstart = () => {
      indicator.innerText = 'loading...'
    }

    const loadstop = () => {
      indicator.innerText = ''
    }

    webview.addEventListener('did-start-loading', loadstart)
    webview.addEventListener('did-stop-loading', loadstop)
  }
</script>
```

Internal implementation

Under the hood `webview` is implemented with Out-of-Process iframes (OOPIFs). The `webview` tag is essentially a custom element using shadow DOM to wrap an `iframe` element inside it.

So the behavior of `webview` is very similar to a cross-domain `iframe`, as examples:

- When clicking into a `webview`, the page focus will move from the embedder frame to `webview`.
- You can not add keyboard, mouse, and scroll event listeners to `webview`.
- All reactions between the embedder frame and `webview` are asynchronous.

CSS Styling Notes

Please note that the `webview` tag's style uses `display:flex;` internally to ensure the child `iframe` element fills the full height and width of its `webview` container when used with traditional and flexbox layouts. Please do not overwrite the default `display:flex;` CSS property, unless specifying `display:inline-flex;` for inline layout.

Tag Attributes

The `webview` tag has the following attributes:

`src`

```
<webview src="https://www.github.com/"></webview>
```

A `string` representing the visible URL. Writing to this attribute initiates top-level navigation.

Assigning `src` its own value will reload the current page.

The `src` attribute can also accept data URLs, such as `data:text/plain,Hello, world!`.

`nodeintegration`

```
<webview src="http://www.google.com/" nodeintegration></webview>
```

A `boolean`. When this attribute is present the guest page in `webview` will have node integration and can use node APIs like `require` and `process` to access low level system resources. Node integration is disabled by default in the guest page.

`nodeintegrationinsubframes`

```
<webview src="http://www.google.com/" nodeintegrationinsubframes></webview>
```

A `boolean` for the experimental option for enabling NodeJS support in sub-frames such as iframes inside the `webview`. All your preloads will load for every iframe, you can use `process.isMainFrame` to determine if you are in the main frame or not. This option is disabled by default in the guest page.

`plugins`

```
<webview src="https://www.github.com/" plugins></webview>
```

A `boolean`. When this attribute is present the guest page in `webview` will be able to use browser plugins. Plugins are disabled by default.

`preload`

```
<!-- from a file -->
<webview src="https://www.github.com/" preload="./test.js"></webview>
<!-- or if you want to load from an asar archive -->
<webview src="https://www.github.com/" preload="./app.asar/test.js"></webview>
```

A `string` that specifies a script that will be loaded before other scripts run in the guest page. The protocol of script's URL must be `file:` (even when using `asar:` archives) because it will be loaded by Node's `require` under the hood, which treats `asar:` archives as virtual directories.

When the guest page doesn't have node integration this script will still have access to all Node APIs, but global objects injected by Node will be deleted after this script has finished executing.

`httpreferrer`

```
<webview src="https://www.github.com/" httpreferrer="http://cheng.guru"></webview>
```

A `string` that sets the referrer URL for the guest page.

`useragent`

```
<webview src="https://www.github.com/" useragent="Mozilla/5.0 (Windows NT 6.1;
WOW64; Trident/7.0; AS; rv:11.0) like Gecko"></webview>
```

A `string` that sets the user agent for the guest page before the page is navigated to. Once the page is loaded, use the `setUserAgent` method to change the user agent.

`disablewebsecurity`

```
<webview src="https://www.github.com/" disablewebsecurity></webview>
```

A `boolean`. When this attribute is present the guest page will have web security disabled. Web security is enabled by default.

`partition`

```
<webview src="https://github.com" partition="persist:github"></webview>
<webview src="https://electronjs.org" partition="electron"></webview>
```

A `string` that sets the session used by the page. If `partition` starts with `persist:`, the page will use a persistent session available to all pages in the app with the same `partition`. if there is no `persist:` prefix, the page will use an in-memory session. By assigning the same `partition`, multiple pages can share the same session. If the `partition` is unset then default session of the app will be used.

This value can only be modified before the first navigation, since the session of an active renderer process cannot change. Subsequent attempts to modify the value will fail with a DOM exception.

`allowpopups`

```
<webview src="https://www.github.com/" allowpopups></webview>
```

A `boolean`. When this attribute is present the guest page will be allowed to open new windows. Popups are disabled by default.

`webpreferences`

```
<webview src="https://github.com" webpreferences="allowRunningInsecureContent,
javascript=no"></webview>
```

A `string` which is a comma separated list of strings which specifies the web preferences to be set on the webview. The full list of supported preference strings can be found in BrowserWindow.

The string follows the same format as the features string in `window.open`. A name by itself is given a `true` boolean value. A preference can be set to another value by including an `=`, followed by the value. Special values `yes` and `1` are interpreted as `true`, while `no` and `0` are interpreted as `false`.

`enableblinkfeatures`

```
<webview src="https://www.github.com/" enableblinkfeatures="PreciseMemoryInfo,
CSSVariables"></webview>
```

A `string` which is a list of strings which specifies the blink features to be enabled separated by `,`. The full list of supported feature strings can be found in the RuntimeEnabledFeatures.json5 file.

`disableblinkfeatures`

```
<webview src="https://www.github.com/" disableblinkfeatures="PreciseMemoryInfo,
CSSVariables"></webview>
```

A `string` which is a list of strings which specifies the blink features to be disabled separated by `,`. The full list of supported feature strings can be found in the RuntimeEnabledFeatures.json5 file.

Methods

The `webview` tag has the following methods:

Note: The webview element must be loaded before using the methods.

Example

```
const webview = document.querySelector('webview')
webview.addEventListener('dom-ready', () => {
  webview.openDevTools()
})
```

`<webview>.loadURL(url[, options])`

- `url` URL
- `options` Object (optional)
  - `httpReferrer` (string | Referrer) (optional) - An HTTP Referrer url.
  - `userAgent` string (optional) - A user agent originating the request.
  - `extraHeaders` string (optional) - Extra headers separated by "\n"
  - `postData` (UploadRawData | UploadFile)[] (optional)
  - `baseURLForDataURL` string (optional) - Base url (with trailing path separator) for files to be loaded by the data url. This is needed only if the specified `url` is a data url and needs to load other files.

Returns `Promise<void>` - The promise will resolve when the page has finished loading (see `did-finish-load`), and rejects if the page fails to load (see `did-fail-load`).

Loads the `url` in the webview, the `url` must contain the protocol prefix, e.g. the `http://` or `file://`.

`<webview>.downloadURL(url)`

- `url` string

Initiates a download of the resource at `url` without navigating.

`<webview>.getURL()`

Returns `string` - The URL of guest page.

`<webview>.getTitle()`

Returns `string` - The title of guest page.

`<webview>.isLoading()`

Returns `boolean` - Whether guest page is still loading resources.

`<webview>.isLoadingMainFrame()`

Returns `boolean` - Whether the main frame (and not just iframes or frames within it) is still loading.

`<webview>.isWaitingForResponse()`

Returns `boolean` - Whether the guest page is waiting for a first-response for the main resource of the page.

`<webview>.stop()`

Stops any pending navigation.

`<webview>.reload()`

Reloads the guest page.

`<webview>.reloadIgnoringCache()`

Reloads the guest page and ignores cache.

`<webview>.canGoBack()`

Returns `boolean` - Whether the guest page can go back.

`<webview>.canGoForward()`

Returns `boolean` - Whether the guest page can go forward.

`<webview>.canGoToOffset(offset)`

- `offset` Integer

Returns `boolean` - Whether the guest page can go to `offset`.

`<webview>.clearHistory()`

Clears the navigation history.

`<webview>.goBack()`

Makes the guest page go back.

`<webview>.goForward()`

Makes the guest page go forward.

`<webview>.goToIndex(index)`

- `index` Integer

Navigates to the specified absolute index.

`<webview>.goToOffset(offset)`

- `offset` Integer

Navigates to the specified offset from the "current entry".

`<webview>.isCrashed()`

Returns `boolean` - Whether the renderer process has crashed.

`<webview>.setUserAgent(userAgent)`

- `userAgent` string

Overrides the user agent for the guest page.

`<webview>.getUserAgent()`

Returns `string` - The user agent for guest page.

`<webview>.insertCSS(css)`

- `css` string

Returns `Promise<string>` - A promise that resolves with a key for the inserted CSS that can later be used to remove the CSS via `<webview>.removeInsertedCSS(key)`.

Injects CSS into the current web page and returns a unique key for the inserted stylesheet.

`<webview>.removeInsertedCSS(key)`

- `key` string

Returns `Promise<void>` - Resolves if the removal was successful.

Removes the inserted CSS from the current web page. The stylesheet is identified by its key, which is returned from `<webview>.insertCSS(css)`.

`<webview>.executeJavaScript(code[, userGesture])`

- `code` string
- `userGesture` boolean (optional) - Default `false`.

Returns `Promise<any>` - A promise that resolves with the result of the executed code or is rejected if the result of the code is a rejected promise.

Evaluates `code` in page. If `userGesture` is set, it will create the user gesture context in the page. HTML APIs like `requestFullScreen`, which require user action, can take advantage of this option for automation.

`<webview>.openDevTools()`

Opens a DevTools window for guest page.

`<webview>.closeDevTools()`

Closes the DevTools window of guest page.

`<webview>.isDevToolsOpened()`

Returns `boolean` - Whether guest page has a DevTools window attached.

`<webview>.isDevToolsFocused()`

Returns `boolean` - Whether DevTools window of guest page is focused.

`<webview>.inspectElement(x, y)`

- `x` Integer
- `y` Integer

Starts inspecting element at position (`x` , `y`) of guest page.

`<webview>.inspectSharedWorker()`

Opens the DevTools for the shared worker context present in the guest page.

`<webview>.inspectServiceWorker()`

Opens the DevTools for the service worker context present in the guest page.

`<webview>.setAudioMuted(muted)`

- `muted` boolean

Set guest page muted.

`<webview>.isAudioMuted()`

Returns `boolean` - Whether guest page has been muted.

`<webview>.isCurrentlyAudible()`

Returns `boolean` - Whether audio is currently playing.

`<webview>.undo()`

Executes editing command `undo` in page.

`<webview>.redo()`

Executes editing command `redo` in page.

`<webview>.cut()`

Executes editing command `cut` in page.

`<webview>.copy()`

Executes editing command `copy` in page.

`<webview>.paste()`

Executes editing command `paste` in page.

`<webview>.pasteAndMatchStyle()`

Executes editing command `pasteAndMatchStyle` in page.

`<webview>.delete()`

Executes editing command `delete` in page.

`<webview>.selectAll()`

Executes editing command `selectAll` in page.

`<webview>.unselect()`

Executes editing command `unselect` in page.

`<webview>.replace(text)`

- `text` string

Executes editing command `replace` in page.

`<webview>.replaceMisspelling(text)`

- `text` string

Executes editing command `replaceMisspelling` in page.

`<webview>.insertText(text)`

- `text` string

Returns `Promise<void>`

Inserts `text` to the focused element.

`<webview>.findInPage(text[, options])`

- `text` string - Content to be searched, must not be empty.
- `options` Object (optional)
  - `forward` boolean (optional) - Whether to search forward or backward, defaults to `true`.
  - `findNext` boolean (optional) - Whether to begin a new text finding session with this request. Should be `true` for initial requests, and `false` for follow-up requests. Defaults to `false`.
  - `matchCase` boolean (optional) - Whether search should be case-sensitive, defaults to `false`.

Returns `Integer` - The request id used for the request.

Starts a request to find all matches for the `text` in the web page. The result of the request can be obtained by subscribing to `found-in-page` event.

`<webview>.stopFindInPage(action)`

- `action` string - Specifies the action to take place when ending `<webview>.findInPage` request.
  - `clearSelection` - Clear the selection.
  - `keepSelection` - Translate the selection into a normal selection.
  - `activateSelection` - Focus and click the selection node.

Stops any `findInPage` request for the `webview` with the provided `action`.

`<webview>.print([options])`

- `options` Object (optional)
  - `silent` boolean (optional) - Don't ask user for print settings. Default is `false`.
  - `printBackground` boolean (optional) - Prints the background color and image of the web page. Default is `false`.
  - `deviceName` string (optional) - Set the printer device name to use. Must be the system-defined name and not the 'friendly' name, e.g 'Brother_QL_820NWB' and not 'Brother QL-820NWB'.
  - `color` boolean (optional) - Set whether the printed web page will be in color or grayscale. Default is `true`.
  - `margins` Object (optional)
    - `marginType` string (optional) - Can be `default`, `none`, `printableArea`, or `custom`. If `custom` is chosen, you will also need to specify `top`, `bottom`, `left`, and `right`.
    - `top` number (optional) - The top margin of the printed web page, in pixels.
    - `bottom` number (optional) - The bottom margin of the printed web page, in pixels.
    - `left` number (optional) - The left margin of the printed web page, in pixels.
    - `right` number (optional) - The right margin of the printed web page, in pixels.
  - `landscape` boolean (optional) - Whether the web page should be printed in landscape mode. Default is `false`.
  - `scaleFactor` number (optional) - The scale factor of the web page.
  - `pagesPerSheet` number (optional) - The number of pages to print per page sheet.
  - `collate` boolean (optional) - Whether the web page should be collated.
  - `copies` number (optional) - The number of copies of the web page to print.
  - `pageRanges` Object[] (optional) - The page range to print.
    - `from` number - Index of the first page to print (0-based).
    - `to` number - Index of the last page to print (inclusive) (0-based).
  - `duplexMode` string (optional) - Set the duplex mode of the printed web page. Can be `simplex`, `shortEdge`, or `longEdge`.
  - `dpi` Record<string, number> (optional)
    - `horizontal` number (optional) - The horizontal dpi.
    - `vertical` number (optional) - The vertical dpi.
  - `header` string (optional) - string to be printed as page header.
  - `footer` string (optional) - string to be printed as page footer.
  - `pageSize` string | Size (optional) - Specify page size of the printed document. Can be `A3`, `A4`, `A5`, `Legal`, `Letter`, `Tabloid` or an Object containing `height`.

Returns `Promise<void>`

Prints `webview`'s web page. Same as `webContents.print([options])`.

`<webview>.printToPDF(options)`

- `options` Object
  - `headerFooter` Record<string, string> (optional) - the header and footer for the PDF.
    - `title` string - The title for the PDF header.
    - `url` string - the url for the PDF footer.
  - `landscape` boolean (optional) - `true` for landscape, `false` for portrait.
  - `marginsType` Integer (optional) - Specifies the type of margins to use. Uses 0 for default margin, 1 for no margin, and 2 for minimum margin. and `width` in microns.
  - `scaleFactor` number (optional) - The scale factor of the web page. Can range from 0 to 100.
  - `pageRanges` Record<string, number> (optional) - The page range to print. On macOS, only the first range is honored.
    - `from` number - Index of the first page to print (0-based).
    - `to` number - Index of the last page to print (inclusive) (0-based).
  - `pageSize` string | Size (optional) - Specify page size of the generated PDF. Can be `A3`, `A4`, `A5`, `Legal`, `Letter`, `Tabloid` or an Object containing `height`
  - `printBackground` boolean (optional) - Whether to print CSS backgrounds.
  - `printSelectionOnly` boolean (optional) - Whether to print selection only.

Returns `Promise<Uint8Array>` - Resolves with the generated PDF data.

Prints `webview` 's web page as PDF, Same as `webContents.printToPDF(options)`.

`<webview>.capturePage([rect])`

- `rect` Rectangle (optional) - The area of the page to be captured.

Returns `Promise<NativeImage>` - Resolves with a NativeImage

Captures a snapshot of the page within `rect`. Omitting `rect` will capture the whole visible page.

`<webview>.send(channel, ...args)`

- `channel` string
- `...args` any[]

Returns `Promise<void>`

Send an asynchronous message to renderer process via `channel`, you can also send arbitrary arguments. The renderer process can handle the message by listening to the `channel` event with the `ipcRenderer` module.

See webContents.send for examples.

`<webview>.sendToFrame(frameId, channel, ...args)`

- `frameId` [number, number] - `[processId, frameId]`
- `channel` string
- `...args` any[]

Returns `Promise<void>`

Send an asynchronous message to renderer process via `channel`, you can also send arbitrary arguments. The renderer process can handle the message by listening to the `channel` event with the `ipcRenderer` module.

See webContents.sendToFrame for examples.

`<webview>.sendInputEvent(event)`

- `event` MouseInputEvent | MouseWheelInputEvent | KeyboardInputEvent

Returns `Promise<void>`

Sends an input `event` to the page.

See webContents.sendInputEvent for detailed description of `event` object.

`<webview>.setZoomFactor(factor)`

- `factor` number - Zoom factor.

Changes the zoom factor to the specified factor. Zoom factor is zoom percent divided by 100, so 300% = 3.0.

`<webview>.setZoomLevel(level)`

- `level` number - Zoom level.

Changes the zoom level to the specified level. The original size is 0 and each increment above or below represents zooming 20% larger or smaller to default limits of 300% and 50% of original size, respectively. The formula for this is `scale := 1.2 ^ level`.

> ***NOTE**: The zoom policy at the Chromium level is same-origin, meaning that the zoom level for a specific domain propagates across all instances of windows with the same domain. Differentiating the window URLs will make zoom work per-window.*

`<webview>.getZoomFactor()`

Returns `number` - the current zoom factor.

`<webview>.getZoomLevel()`

Returns `number` - the current zoom level.

`<webview>.setVisualZoomLevelLimits(minimumLevel, maximumLevel)`

- `minimumLevel` number
- `maximumLevel` number

Returns `Promise<void>`

Sets the maximum and minimum pinch-to-zoom level.

`<webview>.showDdefinitionForSelection()` *macOS*

Shows pop-up dictionary that searches the selected word on the page.

`<webview>.getWebContentsId()`

Returns `number` - The WebContents ID of this `webview`.

DOM Events

The following DOM events are available to the `webview` tag:

Event: 'load-commit'

Returns:

- `url` string
- `isMainFrame` boolean

Fired when a load has committed. This includes navigation within the current document as well as subframe document-level loads, but does not include asynchronous resource loads.

Event: 'did-finish-load'

Fired when the navigation is done, i.e. the spinner of the tab will stop spinning, and the `onload` event is dispatched.

Event: 'did-fail-load'

Returns:

- `errorCode` Integer
- `errorDescription` string
- `validatedURL` string
- `isMainFrame` boolean

This event is like `did-finish-load`, but fired when the load failed or was cancelled, e.g. `window.stop()` is invoked.

Event: 'did-frame-finish-load'

Returns:

- `isMainFrame` boolean

Fired when a frame has done navigation.

Event: 'did-start-loading'

Corresponds to the points in time when the spinner of the tab starts spinning.

Event: 'did-stop-loading'

Corresponds to the points in time when the spinner of the tab stops spinning.

Event: 'did-attach'

Fired when attached to the embedder web contents.

Event: 'dom-ready'

Fired when document in the given frame is loaded.

Event: 'page-title-updated'

Returns:

- `title` string
- `explicitSet` boolean

Fired when page title is set during navigation. `explicitSet` is false when title is synthesized from file url.

Event: 'page-favicon-updated'

Returns:

- `favicons` string[] - Array of URLs.

Fired when page receives favicon urls.

Event: 'enter-html-full-screen'

Fired when page enters fullscreen triggered by HTML API.

Event: 'leave-html-full-screen'

Fired when page leaves fullscreen triggered by HTML API.

Event: 'console-message'

Returns:

- `level` Integer - The log level, from 0 to 3. In order it matches `verbose`, `info`, `warning` and `error`.
- `message` string - The actual console message
- `line` Integer - The line number of the source that triggered this console message
- `sourceId` string

Fired when the guest window logs a console message.

The following example code forwards all log messages to the embedder's console without regard for log level or other properties.

```
const webview = document.querySelector('webview')
webview.addEventListener('console-message', (e) => {
  console.log('Guest page logged a message:', e.message)
})
```

Event: 'found-in-page'

Returns:

- `result` Object
  - `requestId` Integer
  - `activeMatchOrdinal` Integer - Position of the active match.
  - `matches` Integer - Number of Matches.
  - `selectionArea` Rectangle - Coordinates of first match region.
  - `finalUpdate` boolean

Fired when a result is available for `webview.findInPage` request.

```
const webview = document.querySelector('webview')
webview.addEventListener('found-in-page', (e) => {
  webview.stopFindInPage('keepSelection')
})

const requestId = webview.findInPage('test')
console.log(requestId)
```

Event: 'new-window'

Returns:

- `url` string
- `frameName` string
- `disposition` string - Can be `default`, `foreground-tab`, `background-tab`, `new-window`, `save-to-disk` and `other`.
- `options` BrowserWindowConstructorOptions - The options which should be used for creating the new `BrowserWindow`.

Fired when the guest page attempts to open a new browser window.

The following example code opens the new url in system's default browser.

```
const { shell } = require('electron')
const webview = document.querySelector('webview')

webview.addEventListener('new-window', async (e) => {
  const protocol = (new URL(e.url)).protocol
  if (protocol === 'http:' || protocol === 'https:') {
    await shell.openExternal(e.url)
  }
})
```

Event: 'will-navigate'

Returns:

- `url` string

Emitted when a user or the page wants to start navigation. It can happen when the `window.location` object is changed or a user clicks a link in the page.

This event will not emit when the navigation is started programmatically with APIs like `<webview>.loadURL` and `<webview>.back`.

It is also not emitted during in-page navigation, such as clicking anchor links or updating the `window.location.hash`. Use `did-navigate-in-page` event for this purpose.

Calling `event.preventDefault()` does **NOT** have any effect.

Event: 'did-start-navigation'

Returns:

- `url` string
- `isInPlace` boolean
- `isMainFrame` boolean
- `frameProcessId` Integer
- `frameRoutingId` Integer

Emitted when any frame (including main) starts navigating. `isInPlace` will be `true` for in-page navigations.

Event: 'did-redirect-navigation'

Returns:

- `url` string
- `isInPlace` boolean
- `isMainFrame` boolean
- `frameProcessId` Integer
- `frameRoutingId` Integer

Emitted after a server side redirect occurs during navigation. For example a 302 redirect.

Event: 'did-navigate'

Returns:

- `url` string

Emitted when a navigation is done.

This event is not emitted for in-page navigations, such as clicking anchor links or updating the `window.location.hash`. Use `did-navigate-in-page` event for this purpose.

Event: 'did-frame-navigate'

Returns:

- `url` string
- `httpResponseCode` Integer - -1 for non HTTP navigations
- `httpStatusText` string - empty for non HTTP navigations,
- `isMainFrame` boolean
- `frameProcessId` Integer
- `frameRoutingId` Integer

Emitted when any frame navigation is done.

This event is not emitted for in-page navigations, such as clicking anchor links or updating the `window.location.hash`. Use `did-navigate-in-page` event for this purpose.

Event: 'did-navigate-in-page'

Returns:

- `isMainFrame` boolean
- `url` string

Emitted when an in-page navigation happened.

When in-page navigation happens, the page URL changes but does not cause navigation outside of the page. Examples of this occurring are when anchor links are clicked or when the DOM `hashchange` event is triggered.

Event: 'close'

Fired when the guest page attempts to close itself.

The following example code navigates the `webview` to `about:blank` when the guest attempts to close itself.

```
const webview = document.querySelector('webview')
webview.addEventListener('close', () => {
  webview.src = 'about:blank'
})
```

Event: 'ipc-message'

Returns:

- `frameId` [number, number] - pair of `[processId, frameId]`.
- `channel` string
- `args` any[]

Fired when the guest page has sent an asynchronous message to embedder page.

With `sendToHost` method and `ipc-message` event you can communicate between guest page and embedder page:

```
// In embedder page.
const webview = document.querySelector('webview')
webview.addEventListener('ipc-message', (event) => {
  console.log(event.channel)
  // Prints "pong"
})
webview.send('ping')
```

```
// In guest page.
const { ipcRenderer } = require('electron')
ipcRenderer.on('ping', () => {
  ipcRenderer.sendToHost('pong')
})
```

Event: 'crashed'

Fired when the renderer process is crashed.

Event: 'plugin-crashed'

Returns:

- `name` string
- `version` string

Fired when a plugin process is crashed.

Event: 'destroyed'

Fired when the WebContents is destroyed.

Event: 'media-started-playing'

Emitted when media starts playing.

Event: 'media-paused'

Emitted when media is paused or done playing.

Event: 'did-change-theme-color'

Returns:

- `themeColor` string

Emitted when a page's theme color changes. This is usually due to encountering a meta tag:

```
<meta name='theme-color' content='#ff0000'>
```

Event: 'update-target-url'

Returns:

- `url` string

Emitted when mouse moves over a link or the keyboard moves the focus to a link.

Event: 'devtools-opened'

Emitted when DevTools is opened.

Event: 'devtools-closed'

Emitted when DevTools is closed.

Event: 'devtools-focused'

Emitted when DevTools is focused / opened.

Event: 'context-menu'

Returns:

- `params` Object
  - `x` Integer - x coordinate.
  - `y` Integer - y coordinate.
  - `linkURL` string - URL of the link that encloses the node the context menu was invoked on.
  - `linkText` string - Text associated with the link. May be an empty string if the contents of the link are an image.
  - `pageURL` string - URL of the top level page that the context menu was invoked on.
  - `frameURL` string - URL of the subframe that the context menu was invoked on.

- `srcURL` string - Source URL for the element that the context menu was invoked on. Elements with source URLs are images, audio and video.
- `mediaType` string - Type of the node the context menu was invoked on. Can be `none`, `image`, `audio`, `video`, `canvas`, `file` or `plugin`.
- `hasImageContents` boolean - Whether the context menu was invoked on an image which has non-empty contents.
- `isEditable` boolean - Whether the context is editable.
- `selectionText` string - Text of the selection that the context menu was invoked on.
- `titleText` string - Title text of the selection that the context menu was invoked on.
- `altText` string - Alt text of the selection that the context menu was invoked on.
- `suggestedFilename` string - Suggested filename to be used when saving file through 'Save Link As' option of context menu.
- `selectionRect` Rectangle - Rect representing the coordinates in the document space of the selection.
- `selectionStartOffset` number - Start position of the selection text.
- `referrerPolicy` Referrer - The referrer policy of the frame on which the menu is invoked.
- `misspelledWord` string - The misspelled word under the cursor, if any.
- `dictionarySuggestions` string[] - An array of suggested words to show the user to replace the `misspelledWord`. Only available if there is a misspelled word and spellchecker is enabled.
- `frameCharset` string - The character encoding of the frame on which the menu was invoked.
- `inputFieldType` string - If the context menu was invoked on an input field, the type of that field. Possible values are `none`, `plainText`, `password`, `other`.
- `spellcheckEnabled` boolean - If the context is editable, whether or not spellchecking is enabled.
- `menuSourceType` string - Input source that invoked the context menu. Can be `none`, `mouse`, `keyboard`, `touch`, `touchMenu`, `longPress`, `longTap`, `touchHandle`, `stylus`, `adjustSelection`, or `adjustSelectionReset`.
- `mediaFlags` Object - The flags for the media element the context menu was invoked on.
  - `inError` boolean - Whether the media element has crashed.
  - `isPaused` boolean - Whether the media element is paused.
  - `isMuted` boolean - Whether the media element is muted.
  - `hasAudio` boolean - Whether the media element has audio.
  - `isLooping` boolean - Whether the media element is looping.
  - `isControlsVisible` boolean - Whether the media element's controls are visible.
  - `canToggleControls` boolean - Whether the media element's controls are toggleable.
  - `canPrint` boolean - Whether the media element can be printed.
  - `canSave` boolean - Whether or not the media element can be downloaded.
  - `canShowPictureInPicture` boolean - Whether the media element can show picture-in-picture.
  - `isShowingPictureInPicture` boolean - Whether the media element is currently showing picture-in-picture.
  - `canRotate` boolean - Whether the media element can be rotated.
  - `canLoop` boolean - Whether the media element can be looped.
- `editFlags` Object - These flags indicate whether the renderer believes it is able to perform the corresponding action.
  - `canUndo` boolean - Whether the renderer believes it can undo.
  - `canRedo` boolean - Whether the renderer believes it can redo.

- `canCut` boolean - Whether the renderer believes it can cut.
- `canCopy` boolean - Whether the renderer believes it can copy.
- `canPaste` boolean - Whether the renderer believes it can paste.
- `canDelete` boolean - Whether the renderer believes it can delete.
- `canSelectAll` boolean - Whether the renderer believes it can select all.
- `canEditRichly` boolean - Whether the renderer believes it can edit text richly.

Emitted when there is a new context menu that needs to be handled.