BrowserWindow

> *Create and control browser windows.*

Process: Main

```
// In the main process.
const { BrowserWindow } = require('electron')

const win = new BrowserWindow({ width: 800, height: 600 })

// Load a remote URL
win.loadURL('https://github.com')

// Or load a local HTML file
win.loadFile('index.html')
```

Window customization

The `BrowserWindow` class exposes various ways to modify the look and behavior of your app's windows. For more details, see the Window Customization tutorial.

Showing the window gracefully

When loading a page in the window directly, users may see the page load incrementally, which is not a good experience for a native app. To make the window display without a visual flash, there are two solutions for different situations.

Using the `ready-to-show` event

While loading the page, the `ready-to-show` event will be emitted when the renderer process has rendered the page for the first time if the window has not been shown yet. Showing the window after this event will have no visual flash:

```
const { BrowserWindow } = require('electron')
const win = new BrowserWindow({ show: false })
win.once('ready-to-show', () => {
  win.show()
})
```

This event is usually emitted after the `did-finish-load` event, but for pages with many remote resources, it may be emitted before the `did-finish-load` event.

Please note that using this event implies that the renderer will be considered "visible" and paint even though `show` is false. This event will never fire if you use `paintWhenInitiallyHidden: false`

Setting the `backgroundColor` property

For a complex app, the `ready-to-show` event could be emitted too late, making the app feel slow. In this case, it is recommended to show the window immediately, and use a `backgroundColor` close to your app's background:

```
const { BrowserWindow } = require('electron')

const win = new BrowserWindow({ backgroundColor: '#2e2c29' })
win.loadURL('https://github.com')
```

Note that even for apps that use `ready-to-show` event, it is still recommended to set `backgroundColor` to make the app feel more native.

Some examples of valid `backgroundColor` values include:

```
const win = new BrowserWindow()
win.setBackgroundColor('hsl(230, 100%, 50%)')
win.setBackgroundColor('rgb(255, 145, 145)')
win.setBackgroundColor('#ff00a3')
win.setBackgroundColor('blueviolet')
```

For more information about these color types see valid options in win.setBackgroundColor.

Parent and child windows

By using `parent` option, you can create child windows:

```
const { BrowserWindow } = require('electron')

const top = new BrowserWindow()
const child = new BrowserWindow({ parent: top })
child.show()
top.show()
```

The `child` window will always show on top of the `top` window.

Modal windows

A modal window is a child window that disables parent window, to create a modal window, you have to set both `parent` and `modal` options:

```
const { BrowserWindow } = require('electron')

const child = new BrowserWindow({ parent: top, modal: true, show: false })
child.loadURL('https://github.com')
child.once('ready-to-show', () => {
  child.show()
})
```

Page visibility

The Page Visibility API works as follows:

- On all platforms, the visibility state tracks whether the window is hidden/minimized or not.

- Additionally, on macOS, the visibility state also tracks the window occlusion state. If the window is occluded (i.e. fully covered) by another window, the visibility state will be `hidden`. On other platforms, the visibility state will be `hidden` only when the window is minimized or explicitly hidden with `win.hide()`.
- If a `BrowserWindow` is created with `show: false`, the initial visibility state will be `visible` despite the window actually being hidden.
- If `backgroundThrottling` is disabled, the visibility state will remain `visible` even if the window is minimized, occluded, or hidden.

It is recommended that you pause expensive operations when the visibility state is `hidden` in order to minimize power consumption.

Platform notices

- On macOS modal windows will be displayed as sheets attached to the parent window.
- On macOS the child windows will keep the relative position to parent window when parent window moves, while on Windows and Linux child windows will not move.
- On Linux the type of modal windows will be changed to `dialog`.
- On Linux many desktop environments do not support hiding a modal window.

Class: BrowserWindow

> *Create and control browser windows.*

Process: [Main]

`BrowserWindow` is an [EventEmitter].

It creates a new `BrowserWindow` with native properties as set by the `options`.

`new BrowserWindow([options])`

- `options` Object (optional)
  - `width` Integer (optional) - Window's width in pixels. Default is `800`.
  - `height` Integer (optional) - Window's height in pixels. Default is `600`.
  - `x` Integer (optional) - (**required** if y is used) Window's left offset from screen. Default is to center the window.
  - `y` Integer (optional) - (**required** if x is used) Window's top offset from screen. Default is to center the window.
  - `useContentSize` boolean (optional) - The `width` and `height` would be used as web page's size, which means the actual window's size will include window frame's size and be slightly larger. Default is `false`.
  - `center` boolean (optional) - Show window in the center of the screen. Default is `false`.
  - `minWidth` Integer (optional) - Window's minimum width. Default is `0`.
  - `minHeight` Integer (optional) - Window's minimum height. Default is `0`.
  - `maxWidth` Integer (optional) - Window's maximum width. Default is no limit.
  - `maxHeight` Integer (optional) - Window's maximum height. Default is no limit.
  - `resizable` boolean (optional) - Whether window is resizable. Default is `true`.
  - `movable` boolean (optional) *macOS Windows* - Whether window is movable. This is not implemented on Linux. Default is `true`.
  - `minimizable` boolean (optional) *macOS Windows* - Whether window is minimizable. This is not implemented on Linux. Default is `true`.

  - `maximizable` boolean (optional) *macOS Windows* - Whether window is maximizable. This is not implemented on Linux. Default is `true` .
  - `closable` boolean (optional) *macOS Windows* - Whether window is closable. This is not implemented on Linux. Default is `true` .
  - `focusable` boolean (optional) - Whether the window can be focused. Default is `true` . On Windows setting `focusable: false` also implies setting `skipTaskbar: true` . On Linux setting `focusable: false` makes the window stop interacting with wm, so the window will always stay on top in all workspaces.
  - `alwaysOnTop` boolean (optional) - Whether the window should always stay on top of other windows. Default is `false` .
  - `fullscreen` boolean (optional) - Whether the window should show in fullscreen. When explicitly set to `false` the fullscreen button will be hidden or disabled on macOS. Default is `false` .
  - `fullscreenable` boolean (optional) - Whether the window can be put into fullscreen mode. On macOS, also whether the maximize/zoom button should toggle full screen mode or maximize window. Default is `true` .
  - `simpleFullscreen` boolean (optional) *macOS* - Use pre-Lion fullscreen on macOS. Default is `false` .
  - `skipTaskbar` boolean (optional) *macOS Windows* - Whether to show the window in taskbar. Default is `false` .
  - `kiosk` boolean (optional) - Whether the window is in kiosk mode. Default is `false` .
  - `title` string (optional) - Default window title. Default is `"Electron"` . If the HTML tag `<title>` is defined in the HTML file loaded by `loadURL()` , this property will be ignored.
  - `icon` (NativeImage | string) (optional) - The window icon. On Windows it is recommended to use `ICO` icons to get best visual effects, you can also leave it undefined so the executable's icon will be used.
  - `show` boolean (optional) - Whether window should be shown when created. Default is `true` .
  - `paintWhenInitiallyHidden` boolean (optional) - Whether the renderer should be active when `show` is `false` and it has just been created. In order for `document.visibilityState` to work correctly on first load with `show: false` you should set this to `false` . Setting this to `false` will cause the `ready-to-show` event to not fire. Default is `true` .
  - `frame` boolean (optional) - Specify `false` to create a frameless window. Default is `true` .
  - `parent` BrowserWindow (optional) - Specify parent window. Default is `null` .
  - `modal` boolean (optional) - Whether this is a modal window. This only works when the window is a child window. Default is `false` .
  - `acceptFirstMouse` boolean (optional) *macOS* - Whether clicking an inactive window will also click through to the web contents. Default is `false` on macOS. This option is not configurable on other platforms.
  - `disableAutoHideCursor` boolean (optional) - Whether to hide cursor when typing. Default is `false` .
  - `autoHideMenuBar` boolean (optional) - Auto hide the menu bar unless the `Alt` key is pressed. Default is `false` .
  - `enableLargerThanScreen` boolean (optional) *macOS* - Enable the window to be resized larger than screen. Only relevant for macOS, as other OSes allow larger-than-screen windows by default. Default is `false` .

- `backgroundColor` string (optional) - The window's background color in Hex, RGB, RGBA, HSL, HSLA or named CSS color format. Alpha in #AARRGGBB format is supported if `transparent` is set to `true`. Default is `#FFF` (white). See [win.setBackgroundColor]() for more information.
- `hasShadow` boolean (optional) - Whether window should have a shadow. Default is `true`.
- `opacity` number (optional) *macOS Windows* - Set the initial opacity of the window, between 0.0 (fully transparent) and 1.0 (fully opaque). This is only implemented on Windows and macOS.
- `darkTheme` boolean (optional) - Forces using dark theme for the window, only works on some GTK+3 desktop environments. Default is `false`.
- `transparent` boolean (optional) - Makes the window [transparent](). Default is `false`. On Windows, does not work unless the window is frameless.
- `type` string (optional) - The type of window, default is normal window. See more about this below.
- `visualEffectState` string (optional) *macOS* - Specify how the material appearance should reflect window activity state on macOS. Must be used with the `vibrancy` property. Possible values are:
  - `followWindow` - The backdrop should automatically appear active when the window is active, and inactive when it is not. This is the default.
  - `active` - The backdrop should always appear active.
  - `inactive` - The backdrop should always appear inactive.
- `titleBarStyle` string (optional) *macOS Windows* - The style of window title bar. Default is `default`. Possible values are:
  - `default` - Results in the standard title bar for macOS or Windows respectively.
  - `hidden` - Results in a hidden title bar and a full size content window. On macOS, the window still has the standard window controls (“traffic lights”) in the top left. On Windows, when combined with `titleBarOverlay: true` it will activate the Window Controls Overlay (see `titleBarOverlay` for more information), otherwise no window controls will be shown.
  - `hiddenInset` *macOS* - Only on macOS, results in a hidden title bar with an alternative look where the traffic light buttons are slightly more inset from the window edge.
  - `customButtonsOnHover` *macOS* - Only on macOS, results in a hidden title bar and a full size content window, the traffic light buttons will display when being hovered over in the top left of the window. **Note:** This option is currently experimental.
- `trafficLightPosition` [Point]() (optional) *macOS* - Set a custom position for the traffic light buttons in frameless windows.
- `roundedCorners` boolean (optional) *macOS* - Whether frameless window should have rounded corners on macOS. Default is `true`.
- `fullscreenWindowTitle` boolean (optional) *macOS Deprecated* - Shows the title in the title bar in full screen mode on macOS for `hiddenInset` titleBarStyle. Default is `false`.
- `thickFrame` boolean (optional) - Use `WS_THICKFRAME` style for frameless windows on Windows, which adds standard window frame. Setting it to `false` will remove window shadow and window animations. Default is `true`.
- `vibrancy` string (optional) *macOS* - Add a type of vibrancy effect to the window, only on macOS. Can be `appearance-based`, `light`, `dark`, `titlebar`, `selection`, `menu`, `popover`, `sidebar`, `medium-light`, `ultra-dark`, `header`, `sheet`, `window`, `hud`, `fullscreen-ui`, `tooltip`, `content`, `under-window`, or `under-page`. Please note that `appearance-based`, `light`, `dark`, `medium-light`, and `ultra-dark` are deprecated and have been removed in macOS Catalina (10.15).

- `zoomToPageWidth` boolean (optional) *macOS* - Controls the behavior on macOS when option-clicking the green stoplight button on the toolbar or by clicking the Window > Zoom menu item. If `true`, the window will grow to the preferred width of the web page when zoomed, `false` will cause it to zoom to the width of the screen. This will also affect the behavior when calling `maximize()` directly. Default is `false`.
- `tabbingIdentifier` string (optional) *macOS* - Tab group name, allows opening the window as a native tab on macOS 10.12+. Windows with the same tabbing identifier will be grouped together. This also adds a native new tab button to your window's tab bar and allows your `app` and window to receive the `new-window-for-tab` event.
- `webPreferences` Object (optional) - Settings of web page's features.
  - `devTools` boolean (optional) - Whether to enable DevTools. If it is set to `false`, can not use `BrowserWindow.webContents.openDevTools()` to open DevTools. Default is `true`.
  - `nodeIntegration` boolean (optional) - Whether node integration is enabled. Default is `false`.
  - `nodeIntegrationInWorker` boolean (optional) - Whether node integration is enabled in web workers. Default is `false`. More about this can be found in Multithreading.
  - `nodeIntegrationInSubFrames` boolean (optional) - Experimental option for enabling Node.js support in sub-frames such as iframes and child windows. All your preloads will load for every iframe, you can use `process.isMainFrame` to determine if you are in the main frame or not.
  - `preload` string (optional) - Specifies a script that will be loaded before other scripts run in the page. This script will always have access to node APIs no matter whether node integration is turned on or off. The value should be the absolute file path to the script. When node integration is turned off, the preload script can reintroduce Node global symbols back to the global scope. See example here.
  - `sandbox` boolean (optional) - If set, this will sandbox the renderer associated with the window, making it compatible with the Chromium OS-level sandbox and disabling the Node.js engine. This is not the same as the `nodeIntegration` option and the APIs available to the preload script are more limited. Read more about the option here.
  - `session` Session (optional) - Sets the session used by the page. Instead of passing the Session object directly, you can also choose to use the `partition` option instead, which accepts a partition string. When both `session` and `partition` are provided, `session` will be preferred. Default is the default session.
  - `partition` string (optional) - Sets the session used by the page according to the session's partition string. 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. Default is the default session.
  - `zoomFactor` number (optional) - The default zoom factor of the page, `3.0` represents `300%`. Default is `1.0`.
  - `javascript` boolean (optional) - Enables JavaScript support. Default is `true`.
  - `webSecurity` boolean (optional) - When `false`, it will disable the same-origin policy (usually using testing websites by people), and set `allowRunningInsecureContent` to `true` if this options has not been set by user. Default is `true`.

- `allowRunningInsecureContent` boolean (optional) - Allow an https page to run JavaScript, CSS or plugins from http URLs. Default is `false`.
- `images` boolean (optional) - Enables image support. Default is `true`.
- `imageAnimationPolicy` string (optional) - Specifies how to run image animations (E.g. GIFs). Can be `animate`, `animateOnce` or `noAnimation`. Default is `animate`.
- `textAreasAreResizable` boolean (optional) - Make TextArea elements resizable. Default is `true`.
- `webgl` boolean (optional) - Enables WebGL support. Default is `true`.
- `plugins` boolean (optional) - Whether plugins should be enabled. Default is `false`.
- `experimentalFeatures` boolean (optional) - Enables Chromium's experimental features. Default is `false`.
- `scrollBounce` boolean (optional) *macOS* - Enables scroll bounce (rubber banding) effect on macOS. Default is `false`.
- `enableBlinkFeatures` string (optional) - A list of feature strings separated by `,`, like `CSSVariables,KeyboardEventKey` to enable. The full list of supported feature strings can be found in the RuntimeEnabledFeatures.json5 file.
- `disableBlinkFeatures` string (optional) - A list of feature strings separated by `,`, like `CSSVariables,KeyboardEventKey` to disable. The full list of supported feature strings can be found in the RuntimeEnabledFeatures.json5 file.
- `defaultFontFamily` Object (optional) - Sets the default font for the font-family.
  - `standard` string (optional) - Defaults to `Times New Roman`.
  - `serif` string (optional) - Defaults to `Times New Roman`.
  - `sansSerif` string (optional) - Defaults to `Arial`.
  - `monospace` string (optional) - Defaults to `Courier New`.
  - `cursive` string (optional) - Defaults to `Script`.
  - `fantasy` string (optional) - Defaults to `Impact`.
- `defaultFontSize` Integer (optional) - Defaults to `16`.
- `defaultMonospaceFontSize` Integer (optional) - Defaults to `13`.
- `minimumFontSize` Integer (optional) - Defaults to `0`.
- `defaultEncoding` string (optional) - Defaults to `ISO-8859-1`.
- `backgroundThrottling` boolean (optional) - Whether to throttle animations and timers when the page becomes background. This also affects the Page Visibility API. Defaults to `true`.
- `offscreen` boolean (optional) - Whether to enable offscreen rendering for the browser window. Defaults to `false`. See the offscreen rendering tutorial for more details.
- `contextIsolation` boolean (optional) - Whether to run Electron APIs and the specified `preload` script in a separate JavaScript context. Defaults to `true`. The context that the `preload` script runs in will only have access to its own dedicated `document` and `window` globals, as well as its own set of JavaScript builtins (`Array`, `Object`, `JSON`, etc.), which are all invisible to the loaded content. The Electron API will only be available in the `preload` script and not the loaded page. This option should be used when loading potentially untrusted remote content to ensure the loaded content cannot tamper with the `preload` script and any Electron APIs being used. This option uses the same technique used by Chrome Content Scripts. You can access this context in

the dev tools by selecting the 'Electron Isolated Context' entry in the combo box at the top of the Console tab.

    - `webviewTag` boolean (optional) - Whether to enable the [`<webview>` tag](). Defaults to `false`. **Note:** The `preload` script configured for the `<webview>` will have node integration enabled when it is executed so you should ensure remote/untrusted content is not able to create a `<webview>` tag with a possibly malicious `preload` script. You can use the `will-attach-webview` event on [webContents]() to strip away the `preload` script and to validate or alter the `<webview>` 's initial settings.
    - `additionalArguments` string[] (optional) - A list of strings that will be appended to `process.argv` in the renderer process of this app. Useful for passing small bits of data down to renderer process preload scripts.
    - `safeDialogs` boolean (optional) - Whether to enable browser style consecutive dialog protection. Default is `false`.
    - `safeDialogsMessage` string (optional) - The message to display when consecutive dialog protection is triggered. If not defined the default message would be used, note that currently the default message is in English and not localized.
    - `disableDialogs` boolean (optional) - Whether to disable dialogs completely. Overrides `safeDialogs`. Default is `false`.
    - `navigateOnDragDrop` boolean (optional) - Whether dragging and dropping a file or link onto the page causes a navigation. Default is `false`.
    - `autoplayPolicy` string (optional) - Autoplay policy to apply to content in the window, can be `no-user-gesture-required`, `user-gesture-required`, `document-user-activation-required`. Defaults to `no-user-gesture-required`.
    - `disableHtmlFullscreenWindowResize` boolean (optional) - Whether to prevent the window from resizing when entering HTML Fullscreen. Default is `false`.
    - `accessibleTitle` string (optional) - An alternative title string provided only to accessibility tools such as screen readers. This string is not directly visible to users.
    - `spellcheck` boolean (optional) - Whether to enable the builtin spellchecker. Default is `true`.
    - `enableWebSQL` boolean (optional) - Whether to enable the [WebSQL api](). Default is `true`.
    - `v8CacheOptions` string (optional) - Enforces the v8 code caching policy used by blink. Accepted values are
        - `none` - Disables code caching
        - `code` - Heuristic based code caching
        - `bypassHeatCheck` - Bypass code caching heuristics but with lazy compilation
        - `bypassHeatCheckAndEagerCompile` - Same as above except compilation is eager. Default policy is `code`.
    - `enablePreferredSizeMode` boolean (optional) - Whether to enable preferred size mode. The preferred size is the minimum size needed to contain the layout of the document—without requiring scrolling. Enabling this will cause the `preferred-size-changed` event to be emitted on the `WebContents` when the preferred size changes. Default is `false`.
  - `titleBarOverlay` Object | Boolean (optional) - When using a frameless window in conjunction with `win.setWindowButtonVisibility(true)` on macOS or using a `titleBarStyle` so that the standard window controls ("traffic lights" on macOS) are visible, this property enables the

    Window Controls Overlay [JavaScript APIs](#) and [CSS Environment Variables](#). Specifying `true` will result in an overlay with default system colors. Default is `false`.
        - `color` String (optional) *Windows* - The CSS color of the Window Controls Overlay when enabled. Default is the system color.
        - `symbolColor` String (optional) *Windows* - The CSS color of the symbols on the Window Controls Overlay when enabled. Default is the system color.
        - `height` Integer (optional) *macOS Windows* - The height of the title bar and Window Controls Overlay in pixels. Default is system height.

When setting minimum or maximum window size with `minWidth` / `maxWidth` / `minHeight` / `maxHeight`, it only constrains the users. It won't prevent you from passing a size that does not follow size constraints to `setBounds` / `setSize` or to the constructor of `BrowserWindow`.

The possible values and behaviors of the `type` option are platform dependent. Possible values are:

- On Linux, possible types are `desktop`, `dock`, `toolbar`, `splash`, `notification`.
- On macOS, possible types are `desktop`, `textured`.
    - The `textured` type adds metal gradient appearance (`NSTexturedBackgroundWindowMask`).
    - The `desktop` type places the window at the desktop background window level (`kCGDesktopWindowLevel - 1`). Note that desktop window will not receive focus, keyboard or mouse events, but you can use `globalShortcut` to receive input sparingly.
- On Windows, possible type is `toolbar`.

Instance Events

Objects created with `new BrowserWindow` emit the following events:

Note: Some events are only available on specific operating systems and are labeled as such.

Event: 'page-title-updated'

Returns:

- `event` Event
- `title` string
- `explicitSet` boolean

Emitted when the document changed its title, calling `event.preventDefault()` will prevent the native window's title from changing. `explicitSet` is false when title is synthesized from file URL.

Event: 'close'

Returns:

- `event` Event

Emitted when the window is going to be closed. It's emitted before the `beforeunload` and `unload` event of the DOM. Calling `event.preventDefault()` will cancel the close.

Usually you would want to use the `beforeunload` handler to decide whether the window should be closed, which will also be called when the window is reloaded. In Electron, returning any value other than `undefined` would cancel the close. For example:

```
window.onbeforeunload = (e) => {
  console.log('I do not want to be closed')

  // Unlike usual browsers that a message box will be prompted to users, returning
  // a non-void value will silently cancel the close.
  // It is recommended to use the dialog API to let the user confirm closing the
  // application.
  e.returnValue = false
}
```

Note: *There is a subtle difference between the behaviors of* `window.onbeforeunload = handler` *and* `window.addEventListener('beforeunload', handler)`. *It is recommended to always set the* `event.returnValue` *explicitly, instead of only returning a value, as the former works more consistently within Electron.*

Event: 'closed'

Emitted when the window is closed. After you have received this event you should remove the reference to the window and avoid using it any more.

Event: 'session-end' ***Windows***

Emitted when window session is going to end due to force shutdown or machine restart or session log off.

Event: 'unresponsive'

Emitted when the web page becomes unresponsive.

Event: 'responsive'

Emitted when the unresponsive web page becomes responsive again.

Event: 'blur'

Emitted when the window loses focus.

Event: 'focus'

Emitted when the window gains focus.

Event: 'show'

Emitted when the window is shown.

Event: 'hide'

Emitted when the window is hidden.

Event: 'ready-to-show'

Emitted when the web page has been rendered (while not being shown) and window can be displayed without a visual flash.

Please note that using this event implies that the renderer will be considered "visible" and paint even though `show` is false. This event will never fire if you use `paintWhenInitiallyHidden: false`

Event: 'maximize'

Emitted when window is maximized.

Event: 'unmaximize'

Emitted when the window exits from a maximized state.

Event: 'minimize'

Emitted when the window is minimized.

Event: 'restore'

Emitted when the window is restored from a minimized state.

Event: 'will-resize' *macOS Windows*

Returns:

- `event` Event
- `newBounds` Rectangle - Size the window is being resized to.
- `details` Object
  - `edge` (string) - The edge of the window being dragged for resizing. Can be `bottom` , `left` , `right` , `top-left` , `top-right` , `bottom-left` or `bottom-right` .

Emitted before the window is resized. Calling `event.preventDefault()` will prevent the window from being resized.

Note that this is only emitted when the window is being resized manually. Resizing the window with `setBounds` / `setSize` will not emit this event.

The possible values and behaviors of the `edge` option are platform dependent. Possible values are:

- On Windows, possible values are `bottom` , `top` , `left` , `right` , `top-left` , `top-right` , `bottom-left` , `bottom-right` .
- On macOS, possible values are `bottom` and `right` .
  - The value `bottom` is used to denote vertical resizing.
  - The value `right` is used to denote horizontal resizing.

Event: 'resize'

Emitted after the window has been resized.

Event: 'resized' *macOS Windows*

Emitted once when the window has finished being resized.

This is usually emitted when the window has been resized manually. On macOS, resizing the window with `setBounds` / `setSize` and setting the `animate` parameter to `true` will also emit this event once resizing has finished.

Event: 'will-move' *macOS Windows*

Returns:

- `event` Event
- `newBounds` Rectangle - Location the window is being moved to.

Emitted before the window is moved. On Windows, calling `event.preventDefault()` will prevent the window from being moved.

Note that this is only emitted when the window is being moved manually. Moving the window with `setPosition` / `setBounds` / `center` will not emit this event.

Event: 'move'

Emitted when the window is being moved to a new position.

Event: 'moved' *macOS Windows*

Emitted once when the window is moved to a new position.

Note: On macOS this event is an alias of `move` .

Event: 'enter-full-screen'

Emitted when the window enters a full-screen state.

Event: 'leave-full-screen'

Emitted when the window leaves a full-screen state.

Event: 'enter-html-full-screen'

Emitted when the window enters a full-screen state triggered by HTML API.

Event: 'leave-html-full-screen'

Emitted when the window leaves a full-screen state triggered by HTML API.

Event: 'always-on-top-changed'

Returns:

- `event` Event
- `isAlwaysOnTop` boolean

Emitted when the window is set or unset to show always on top of other windows.

Event: 'app-command' *Windows Linux*

Returns:

- `event` Event
- `command` string

Emitted when an App Command is invoked. These are typically related to keyboard media keys or browser commands, as well as the "Back" button built into some mice on Windows.

Commands are lowercased, underscores are replaced with hyphens, and the `APPCOMMAND_` prefix is stripped off. e.g. `APPCOMMAND_BROWSER_BACKWARD` is emitted as `browser-backward` .

```
const { BrowserWindow } = require('electron')
const win = new BrowserWindow()
win.on('app-command', (e, cmd) => {
  // Navigate the window back when the user hits their mouse back button
  if (cmd === 'browser-backward' && win.webContents.canGoBack()) {
    win.webContents.goBack()
  }
})
```

The following app commands are explicitly supported on Linux:

- `browser-backward`
- `browser-forward`

Event: 'scroll-touch-begin' *macOS*

Emitted when scroll wheel event phase has begun.

Event: 'scroll-touch-end' *macOS*

Emitted when scroll wheel event phase has ended.

Event: 'scroll-touch-edge' *macOS*

Emitted when scroll wheel event phase filed upon reaching the edge of element.

Event: 'swipe' *macOS*

Returns:

- `event` Event
- `direction` string

Emitted on 3-finger swipe. Possible directions are `up` , `right` , `down` , `left` .

The method underlying this event is built to handle older macOS-style trackpad swiping, where the content on the screen doesn't move with the swipe. Most macOS trackpads are not configured to allow this kind of swiping anymore, so in order for it to emit properly the 'Swipe between pages' preference in `System Preferences > Trackpad > More Gestures` must be set to 'Swipe with two or three fingers'.

Event: 'rotate-gesture' *macOS*

Returns:

- `event` Event
- `rotation` Float

Emitted on trackpad rotation gesture. Continually emitted until rotation gesture is ended. The `rotation` value on each emission is the angle in degrees rotated since the last emission. The last emitted event upon a rotation gesture will always be of value `0` . Counter-clockwise rotation values are positive, while clockwise ones are negative.

Event: 'sheet-begin' *macOS*

Emitted when the window opens a sheet.

Event: 'sheet-end' *macOS*

Emitted when the window has closed a sheet.

Event: 'new-window-for-tab' *macOS*

Emitted when the native new tab button is clicked.

Event: 'system-context-menu' *Windows*

Returns:

- `event` Event
- `point` Point - The screen coordinates the context menu was triggered at

Emitted when the system context menu is triggered on the window, this is normally only triggered when the user right clicks on the non-client area of your window. This is the window titlebar or any area you have declared as `-webkit-app-region: drag` in a frameless window.

Calling `event.preventDefault()` will prevent the menu from being displayed.

Static Methods

The `BrowserWindow` class has the following static methods:

`BrowserWindow.getAllWindows()`

Returns `BrowserWindow[]` - An array of all opened browser windows.

`BrowserWindow.getFocusedWindow()`

Returns `BrowserWindow | null` - The window that is focused in this application, otherwise returns `null`.

`BrowserWindow.fromWebContents(webContents)`

- `webContents` WebContents

Returns `BrowserWindow | null` - The window that owns the given `webContents` or `null` if the contents are not owned by a window.

`BrowserWindow.fromBrowserView(browserView)`

- `browserView` BrowserView

Returns `BrowserWindow | null` - The window that owns the given `browserView`. If the given view is not attached to any window, returns `null`.

`BrowserWindow.fromId(id)`

- `id` Integer

Returns `BrowserWindow | null` - The window with the given `id`.

Instance Properties

Objects created with `new BrowserWindow` have the following properties:

```
const { BrowserWindow } = require('electron')
// In this example `win` is our instance
const win = new BrowserWindow({ width: 800, height: 600 })
win.loadURL('https://github.com')
```

`win.webContents` *Readonly*

A `WebContents` object this window owns. All web page related events and operations will be done via it.

See the `webContents` documentation for its methods and events.

`win.id` *Readonly*

A `Integer` property representing the unique ID of the window. Each ID is unique among all `BrowserWindow` instances of the entire Electron application.

`win.autoHideMenuBar`

A `boolean` property that determines whether the window menu bar should hide itself automatically. Once set, the menu bar will only show when users press the single `Alt` key.

If the menu bar is already visible, setting this property to `true` won't hide it immediately.

`win.simpleFullScreen`

A `boolean` property that determines whether the window is in simple (pre-Lion) fullscreen mode.

`win.fullScreen`

A `boolean` property that determines whether the window is in fullscreen mode.

`win.focusable` *Windows macOS*

A `boolean` property that determines whether the window is focusable.

`win.visibleOnAllWorkspaces` *macOS Linux*

A `boolean` property that determines whether the window is visible on all workspaces.

Note: Always returns false on Windows.

`win.shadow`

A `boolean` property that determines whether the window has a shadow.

`win.menuBarVisible` *Windows Linux*

A `boolean` property that determines whether the menu bar should be visible.

Note: If the menu bar is auto-hide, users can still bring up the menu bar by pressing the single `Alt` key.

`win.kiosk`

A `boolean` property that determines whether the window is in kiosk mode.

`win.documentEdited` *macOS*

A `boolean` property that specifies whether the window’s document has been edited.

The icon in title bar will become gray when set to `true`.

`win.representedFilename` *macOS*

A `string` property that determines the pathname of the file the window represents, and the icon of the file will show in window's title bar.

`win.title`

A `string` property that determines the title of the native window.

Note: The title of the web page can be different from the title of the native window.

`win.minimizable` *macOS Windows*

A `boolean` property that determines whether the window can be manually minimized by user.

On Linux the setter is a no-op, although the getter returns `true`.

`win.maximizable` *macOS Windows*

A `boolean` property that determines whether the window can be manually maximized by user.

On Linux the setter is a no-op, although the getter returns `true`.

`win.fullScreenable`

A `boolean` property that determines whether the maximize/zoom window button toggles fullscreen mode or maximizes the window.

`win.resizable`

A `boolean` property that determines whether the window can be manually resized by user.

`win.closable` *macOS Windows*

A `boolean` property that determines whether the window can be manually closed by user.

On Linux the setter is a no-op, although the getter returns `true`.

`win.movable` *macOS Windows*

A `boolean` property that determines Whether the window can be moved by user.

On Linux the setter is a no-op, although the getter returns `true`.

`win.excludedFromShownWindowsMenu` *macOS*

A `boolean` property that determines whether the window is excluded from the application’s Windows menu. `false` by default.

```
const win = new BrowserWindow({ height: 600, width: 600 })

const template = [
  {
    role: 'windowmenu'
  }
]

win.excludedFromShownWindowsMenu = true

const menu = Menu.buildFromTemplate(template)
Menu.setApplicationMenu(menu)
```

`win.accessibleTitle`

A `string` property that defines an alternative title provided only to accessibility tools such as screen readers. This string is not directly visible to users.

Instance Methods

Objects created with `new BrowserWindow` have the following instance methods:

Note: Some methods are only available on specific operating systems and are labeled as such.

`win.destroy()`

Force closing the window, the `unload` and `beforeunload` event won't be emitted for the web page, and `close` event will also not be emitted for this window, but it guarantees the `closed` event will be emitted.

`win.close()`

Try to close the window. This has the same effect as a user manually clicking the close button of the window. The web page may cancel the close though. See the close event.

`win.focus()`

Focuses on the window.

`win.blur()`

Removes focus from the window.

`win.isFocused()`

Returns `boolean` - Whether the window is focused.

`win.isDestroyed()`

Returns `boolean` - Whether the window is destroyed.

`win.show()`

Shows and gives focus to the window.

`win.showInactive()`

Shows the window but doesn't focus on it.

`win.hide()`

Hides the window.

`win.isVisible()`

Returns `boolean` - Whether the window is visible to the user.

`win.isModal()`

Returns `boolean` - Whether current window is a modal window.

`win.maximize()`

Maximizes the window. This will also show (but not focus) the window if it isn't being displayed already.

`win.unmaximize()`

Unmaximizes the window.

`win.isMaximized()`

Returns `boolean` - Whether the window is maximized.

`win.minimize()`

Minimizes the window. On some platforms the minimized window will be shown in the Dock.

`win.restore()`

Restores the window from minimized state to its previous state.

`win.isMinimized()`

Returns `boolean` - Whether the window is minimized.

`win.setFullScreen(flag)`

- `flag` boolean

Sets whether the window should be in fullscreen mode.

`win.isFullScreen()`

Returns `boolean` - Whether the window is in fullscreen mode.

`win.setSimpleFullScreen(flag)` ***macOS***

- `flag` boolean

Enters or leaves simple fullscreen mode.

Simple fullscreen mode emulates the native fullscreen behavior found in versions of macOS prior to Lion (10.7).

`win.isSimpleFullScreen()` ***macOS***

Returns `boolean` - Whether the window is in simple (pre-Lion) fullscreen mode.

`win.isNormal()`

Returns `boolean` - Whether the window is in normal state (not maximized, not minimized, not in fullscreen mode).

`win.setAspectRatio(aspectRatio[, extraSize])`

- `aspectRatio` Float - The aspect ratio to maintain for some portion of the content view.
- `extraSize` Size (optional) *macOS* - The extra size not to be included while maintaining the aspect ratio.

This will make a window maintain an aspect ratio. The extra size allows a developer to have space, specified in pixels, not included within the aspect ratio calculations. This API already takes into account the difference between a window's size and its content size.

Consider a normal window with an HD video player and associated controls. Perhaps there are 15 pixels of controls on the left edge, 25 pixels of controls on the right edge and 50 pixels of controls below the player. In order to maintain a 16:9 aspect ratio (standard aspect ratio for HD @1920x1080) within the player itself we would call this function with arguments of 16/9 and { width: 40, height: 50 }. The second argument doesn't care where the extra width and height are within the content view--only that they exist. Sum any extra width and height areas you have within the overall content view.

The aspect ratio is not respected when window is resized programmatically with APIs like `win.setSize`.

`win.setBackgroundColor(backgroundColor)`

- `backgroundColor` string - Color in Hex, RGB, RGBA, HSL, HSLA or named CSS color format. The alpha channel is optional for the hex type.

Examples of valid `backgroundColor` values:

- Hex
  - #fff (shorthand RGB)
  - #ffff (shorthand ARGB)
  - #ffffff (RGB)
  - #ffffffff (ARGB)
- RGB
  - rgb(([\d]+),\s*([\d]+),\s*([\d]+))
    - e.g. rgb(255, 255, 255)
- RGBA
  - rgba(([\d]+),\s*([\d]+),\s*([\d]+),\s*([\d.]+))
    - e.g. rgba(255, 255, 255, 1.0)
- HSL
  - hsl((-?[\d.]+),\s*([\d.]+)%,\s*([\d.]+)%)
    - e.g. hsl(200, 20%, 50%)
- HSLA
  - hsla((-?[\d.]+),\s*([\d.]+)%,\s*([\d.]+)%,\s*([\d.]+))
    - e.g. hsla(200, 20%, 50%, 0.5)
- Color name
  - Options are listed in SkParseColor.cpp
  - Similar to CSS Color Module Level 3 keywords, but case-sensitive.
    - e.g. `blueviolet` or `red`

Sets the background color of the window. See Setting `backgroundColor`.

`win.previewFile(path[, displayName])` ***macOS***

- `path` string - The absolute path to the file to preview with QuickLook. This is important as Quick Look uses the file name and file extension on the path to determine the content type of the file to open.
- `displayName` string (optional) - The name of the file to display on the Quick Look modal view. This is purely visual and does not affect the content type of the file. Defaults to `path`.

Uses Quick Look to preview a file at a given path.

`win.closeFilePreview()` ***macOS***

Closes the currently open Quick Look panel.

`win.setBounds(bounds[, animate])`

* `bounds` Partial<Rectangle>
* `animate` boolean (optional) *macOS*

Resizes and moves the window to the supplied bounds. Any properties that are not supplied will default to their current values.

```
const { BrowserWindow } = require('electron')
const win = new BrowserWindow()

// set all bounds properties
win.setBounds({ x: 440, y: 225, width: 800, height: 600 })

// set a single bounds property
win.setBounds({ width: 100 })

// { x: 440, y: 225, width: 100, height: 600 }
console.log(win.getBounds())
```

`win.getBounds()`

Returns `Rectangle` - The `bounds` of the window as `Object`.

`win.getBackgroundColor()`

Returns `string` - Gets the background color of the window in Hex (`#RRGGBB`) format.

See Setting `backgroundColor`.

Note: The alpha value is *not* returned alongside the red, green, and blue values.

`win.setContentBounds(bounds[, animate])`

* `bounds` Rectangle
* `animate` boolean (optional) *macOS*

Resizes and moves the window's client area (e.g. the web page) to the supplied bounds.

`win.getContentBounds()`

Returns `Rectangle` - The `bounds` of the window's client area as `Object`.

`win.getNormalBounds()`

Returns `Rectangle` - Contains the window bounds of the normal state

Note: whatever the current state of the window : maximized, minimized or in fullscreen, this function always returns the position and size of the window in normal state. In normal state, getBounds and getNormalBounds returns the same `Rectangle`.

`win.setEnabled(enable)`

* `enable` boolean

Disable or enable the window.

`win.isEnabled()`

Returns `boolean` - whether the window is enabled.

`win.setSize(width, height[, animate])`

- `width` Integer
- `height` Integer
- `animate` boolean (optional) *macOS*

Resizes the window to `width` and `height`. If `width` or `height` are below any set minimum size constraints the window will snap to its minimum size.

`win.getSize()`

Returns `Integer[]` - Contains the window's width and height.

`win.setContentSize(width, height[, animate])`

- `width` Integer
- `height` Integer
- `animate` boolean (optional) *macOS*

Resizes the window's client area (e.g. the web page) to `width` and `height`.

`win.getContentSize()`

Returns `Integer[]` - Contains the window's client area's width and height.

`win.setMinimumSize(width, height)`

- `width` Integer
- `height` Integer

Sets the minimum size of window to `width` and `height`.

`win.getMinimumSize()`

Returns `Integer[]` - Contains the window's minimum width and height.

`win.setMaximumSize(width, height)`

- `width` Integer
- `height` Integer

Sets the maximum size of window to `width` and `height`.

`win.getMaximumSize()`

Returns `Integer[]` - Contains the window's maximum width and height.

`win.setResizable(resizable)`

- `resizable` boolean

Sets whether the window can be manually resized by the user.

`win.isResizable()`

Returns `boolean` - Whether the window can be manually resized by the user.

`win.setMovable(movable)` ***macOS Windows***

- `movable` boolean

Sets whether the window can be moved by user. On Linux does nothing.

`win.isMovable()` ***macOS Windows***

Returns `boolean` - Whether the window can be moved by user.

On Linux always returns `true` .

`win.setMinimizable(minimizable)` ***macOS Windows***

- `minimizable` boolean

Sets whether the window can be manually minimized by user. On Linux does nothing.

`win.isMinimizable()` ***macOS Windows***

Returns `boolean` - Whether the window can be manually minimized by the user.

On Linux always returns `true` .

`win.setMaximizable(maximizable)` ***macOS Windows***

- `maximizable` boolean

Sets whether the window can be manually maximized by user. On Linux does nothing.

`win.isMaximizable()` ***macOS Windows***

Returns `boolean` - Whether the window can be manually maximized by user.

On Linux always returns `true` .

`win.setFullScreenable(fullscreenable)`

- `fullscreenable` boolean

Sets whether the maximize/zoom window button toggles fullscreen mode or maximizes the window.

`win.isFullScreenable()`

Returns `boolean` - Whether the maximize/zoom window button toggles fullscreen mode or maximizes the window.

`win.setClosable(closable)` ***macOS Windows***

- `closable` boolean

Sets whether the window can be manually closed by user. On Linux does nothing.

`win.isClosable()` ***macOS Windows***

Returns `boolean` - Whether the window can be manually closed by user.

On Linux always returns `true` .

`win.setAlwaysOnTop(flag[, level][, relativeLevel])`

- `flag` boolean
- `level` string (optional) *macOS Windows* - Values include `normal`, `floating`, `torn-off-menu`, `modal-panel`, `main-menu`, `status`, `pop-up-menu`, `screen-saver`, and ~~`dock`~~ (Deprecated). The default is `floating` when `flag` is true. The `level` is reset to `normal` when the flag is false. Note that from `floating` to `status` included, the window is placed below the Dock on macOS and below the taskbar on Windows. From `pop-up-menu` to a higher it is shown above the Dock on macOS and above the taskbar on Windows. See the macOS docs for more details.
- `relativeLevel` Integer (optional) *macOS* - The number of layers higher to set this window relative to the given `level`. The default is `0`. Note that Apple discourages setting levels higher than 1 above `screen-saver`.

Sets whether the window should show always on top of other windows. After setting this, the window is still a normal window, not a toolbox window which can not be focused on.

`win.isAlwaysOnTop()`

Returns `boolean` - Whether the window is always on top of other windows.

`win.moveAbove(mediaSourceId)`

- `mediaSourceId` string - Window id in the format of DesktopCapturerSource's id. For example "window:1869:0".

Moves window above the source window in the sense of z-order. If the `mediaSourceId` is not of type window or if the window does not exist then this method throws an error.

`win.moveTop()`

Moves window to top(z-order) regardless of focus

`win.center()`

Moves window to the center of the screen.

`win.setPosition(x, y[, animate])`

- `x` Integer
- `y` Integer
- `animate` boolean (optional) *macOS*

Moves window to `x` and `y`.

`win.getPosition()`

Returns `Integer[]` - Contains the window's current position.

`win.setTitle(title)`

- `title` string

Changes the title of native window to `title`.

`win.getTitle()`

Returns `string` - The title of the native window.

Note: The title of the web page can be different from the title of the native window.

`win.setSheetOffset(offsetY[, offsetX])` *macOS*

- `offsetY` Float
- `offsetX` Float (optional)

Changes the attachment point for sheets on macOS. By default, sheets are attached just below the window frame, but you may want to display them beneath a HTML-rendered toolbar. For example:

```
const { BrowserWindow } = require('electron')
const win = new BrowserWindow()

const toolbarRect = document.getElementById('toolbar').getBoundingClientRect()
win.setSheetOffset(toolbarRect.height)
```

`win.flashFrame(flag)`

- `flag` boolean

Starts or stops flashing the window to attract user's attention.

`win.setSkipTaskbar(skip)`

- `skip` boolean

Makes the window not show in the taskbar.

`win.setKiosk(flag)`

- `flag` boolean

Enters or leaves kiosk mode.

`win.isKiosk()`

Returns `boolean` - Whether the window is in kiosk mode.

`win.isTabletMode()` *Windows*

Returns `boolean` - Whether the window is in Windows 10 tablet mode.

Since Windows 10 users can [use their PC as tablet](), under this mode apps can choose to optimize their UI for tablets, such as enlarging the titlebar and hiding titlebar buttons.

This API returns whether the window is in tablet mode, and the `resize` event can be be used to listen to changes to tablet mode.

`win.getMediaSourceId()`

Returns `string` - Window id in the format of DesktopCapturerSource's id. For example "window:1324:0".

More precisely the format is `window:id:other_id` where `id` is `HWND` on Windows, `CGWindowID` (`uint64_t`) on macOS and `Window` (`unsigned long`) on Linux. `other_id` is used to identify web contents (tabs) so within the same top level window.

`win.getNativeWindowHandle()`

Returns `Buffer` - The platform-specific handle of the window.

The native type of the handle is `HWND` on Windows, `NSView*` on macOS, and `Window` (`unsigned long`) on Linux.

`win.hookWindowMessage(message, callback)` *Windows*

- `message` Integer
- `callback` Function
  - `wParam` any - The `wParam` provided to the WndProc
  - `lParam` any - The `lParam` provided to the WndProc

Hooks a windows message. The `callback` is called when the message is received in the WndProc.

`win.isWindowMessageHooked(message)` *Windows*

- `message` Integer

Returns `boolean` - `true` or `false` depending on whether the message is hooked.

`win.unhookWindowMessage(message)` *Windows*

- `message` Integer

Unhook the window message.

`win.unhookAllWindowMessages()` *Windows*

Unhooks all of the window messages.

`win.setRepresentedFilename(filename)` *macOS*

- `filename` string

Sets the pathname of the file the window represents, and the icon of the file will show in window's title bar.

`win.getRepresentedFilename()` *macOS*

Returns `string` - The pathname of the file the window represents.

`win.setDocumentEdited(edited)` *macOS*

- `edited` boolean

Specifies whether the window’s document has been edited, and the icon in title bar will become gray when set to `true` .

`win.isDocumentEdited()` *macOS*

Returns `boolean` - Whether the window's document has been edited.

`win.focusOnWebView()`

`win.blurWebView()`

`win.capturePage([rect])`

- `rect` Rectangle (optional) - The bounds to capture

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

Captures a snapshot of the page within `rect`. Omitting `rect` will capture the whole visible page. If the page is not visible, `rect` may be empty.

`win.loadURL(url[, options])`

- `url` string
- `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`).

Same as `webContents.loadURL(url[, options])`.

The `url` can be a remote address (e.g. `http://`) or a path to a local HTML file using the `file://` protocol.

To ensure that file URLs are properly formatted, it is recommended to use Node's `url.format` method:

```
const url = require('url').format({
  protocol: 'file',
  slashes: true,
  pathname: require('path').join(__dirname, 'index.html')
})

win.loadURL(url)
```

You can load a URL using a `POST` request with URL-encoded data by doing the following:

```
win.loadURL('http://localhost:8000/post', {
  postData: [{
    type: 'rawData',
    bytes: Buffer.from('hello=world')
  }],
  extraHeaders: 'Content-Type: application/x-www-form-urlencoded'
})
```

`win.loadFile(filePath[, options])`

- `filePath` string
- `options` Object (optional)
  - `query` Record<string, string> (optional) - Passed to `url.format()`.
  - `search` string (optional) - Passed to `url.format()`.
  - `hash` string (optional) - Passed to `url.format()`.

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`]).

Same as `webContents.loadFile`, `filePath` should be a path to an HTML file relative to the root of your application. See the `webContents` docs for more information.

`win.reload()`

Same as `webContents.reload`.

`win.setMenu(menu)` *Linux Windows*

- `menu` Menu | null

Sets the `menu` as the window's menu bar.

`win.removeMenu()` *Linux Windows*

Remove the window's menu bar.

`win.setProgressBar(progress[, options])`

- `progress` Double
- `options` Object (optional)
  - `mode` string *Windows* - Mode for the progress bar. Can be `none`, `normal`, `indeterminate`, `error` or `paused`.

Sets progress value in progress bar. Valid range is [0, 1.0].

Remove progress bar when progress < 0; Change to indeterminate mode when progress > 1.

On Linux platform, only supports Unity desktop environment, you need to specify the `*.desktop` file name to `desktopName` field in `package.json`. By default, it will assume `{app.name}.desktop`.

On Windows, a mode can be passed. Accepted values are `none`, `normal`, `indeterminate`, `error`, and `paused`. If you call `setProgressBar` without a mode set (but with a value within the valid range), `normal` will be assumed.

`win.setOverlayIcon(overlay, description)` *Windows*

- `overlay` [NativeImage] | null - the icon to display on the bottom right corner of the taskbar icon. If this parameter is `null`, the overlay is cleared
- `description` string - a description that will be provided to Accessibility screen readers

Sets a 16 x 16 pixel overlay onto the current taskbar icon, usually used to convey some sort of application status or to passively notify the user.

`win.setHasShadow(hasShadow)`

- `hasShadow` boolean

Sets whether the window should have a shadow.

`win.hasShadow()`

Returns `boolean` - Whether the window has a shadow.

`win.setOpacity(opacity)` ***Windows macOS***

- `opacity` number - between 0.0 (fully transparent) and 1.0 (fully opaque)

Sets the opacity of the window. On Linux, does nothing. Out of bound number values are clamped to the [0, 1] range.

`win.getOpacity()`

Returns `number` - between 0.0 (fully transparent) and 1.0 (fully opaque). On Linux, always returns 1.

`win.setShape(rects)` ***Windows Linux Experimental***

- `rects` Rectangle[] - Sets a shape on the window. Passing an empty list reverts the window to being rectangular.

Setting a window shape determines the area within the window where the system permits drawing and user interaction. Outside of the given region, no pixels will be drawn and no mouse events will be registered. Mouse events outside of the region will not be received by that window, but will fall through to whatever is behind the window.

`win.setThumbarButtons(buttons)` ***Windows***

- `buttons` ThumbarButton[]

Returns `boolean` - Whether the buttons were added successfully

Add a thumbnail toolbar with a specified set of buttons to the thumbnail image of a window in a taskbar button layout. Returns a `boolean` object indicates whether the thumbnail has been added successfully.

The number of buttons in thumbnail toolbar should be no greater than 7 due to the limited room. Once you setup the thumbnail toolbar, the toolbar cannot be removed due to the platform's limitation. But you can call the API with an empty array to clean the buttons.

The `buttons` is an array of `Button` objects:

- `Button` Object
  - `icon` NativeImage - The icon showing in thumbnail toolbar.
  - `click` Function
  - `tooltip` string (optional) - The text of the button's tooltip.
  - `flags` string[] (optional) - Control specific states and behaviors of the button. By default, it is `['enabled']`.

The `flags` is an array that can include following `string`s:

- `enabled` - The button is active and available to the user.
- `disabled` - The button is disabled. It is present, but has a visual state indicating it will not respond to user action.
- `dismissonclick` - When the button is clicked, the thumbnail window closes immediately.
- `nobackground` - Do not draw a button border, use only the image.
- `hidden` - The button is not shown to the user.
- `noninteractive` - The button is enabled but not interactive; no pressed button state is drawn. This value is intended for instances where the button is used in a notification.

`win.setThumbnailClip(region)` ***Windows***

- `region` Rectangle - Region of the window

Sets the region of the window to show as the thumbnail image displayed when hovering over the window in the taskbar. You can reset the thumbnail to be the entire window by specifying an empty region: `{ x: 0, y: 0, width: 0, height: 0 }`.

`win.setThumbnailToolTip(toolTip)` *Windows*

- `toolTip` string

Sets the toolTip that is displayed when hovering over the window thumbnail in the taskbar.

`win.setAppDetails(options)` *Windows*

- `options` Object
  - `appId` string (optional) - Window's [App User Model ID](). It has to be set, otherwise the other options will have no effect.
  - `appIconPath` string (optional) - Window's [Relaunch Icon]().
  - `appIconIndex` Integer (optional) - Index of the icon in `appIconPath`. Ignored when `appIconPath` is not set. Default is `0`.
  - `relaunchCommand` string (optional) - Window's [Relaunch Command]().
  - `relaunchDisplayName` string (optional) - Window's [Relaunch Display Name]().

Sets the properties for the window's taskbar button.

Note: `relaunchCommand` and `relaunchDisplayName` must always be set together. If one of those properties is not set, then neither will be used.

`win.showDefinitionForSelection()` *macOS*

Same as `webContents.showDefinitionForSelection()`.

`win.setIcon(icon)` *Windows Linux*

- `icon` [NativeImage]() | string

Changes window icon.

`win.setWindowButtonVisibility(visible)` *macOS*

- `visible` boolean

Sets whether the window traffic light buttons should be visible.

`win.setAutoHideMenuBar(hide)` *Windows Linux*

- `hide` boolean

Sets whether the window menu bar should hide itself automatically. Once set the menu bar will only show when users press the single `Alt` key.

If the menu bar is already visible, calling `setAutoHideMenuBar(true)` won't hide it immediately.

`win.isMenuBarAutoHide()` *Windows Linux*

Returns `boolean` - Whether menu bar automatically hides itself.

`win.setMenuBarVisibility(visible)` *Windows Linux*

- `visible` boolean

Sets whether the menu bar should be visible. If the menu bar is auto-hide, users can still bring up the menu bar by pressing the single `Alt` key.

`win.isMenuBarVisible()` *Windows Linux*

Returns `boolean` - Whether the menu bar is visible.

`win.setVisibleOnAllWorkspaces(visible[, options])` *macOS Linux*

- `visible` boolean
- `options` Object (optional)
  - `visibleOnFullScreen` boolean (optional) *macOS* - Sets whether the window should be visible above fullscreen windows.
  - `skipTransformProcessType` boolean (optional) *macOS* - Calling setVisibleOnAllWorkspaces will by default transform the process type between UIElementApplication and ForegroundApplication to ensure the correct behavior. However, this will hide the window and dock for a short time every time it is called. If your window is already of type UIElementApplication, you can bypass this transformation by passing true to skipTransformProcessType.

Sets whether the window should be visible on all workspaces.

Note: This API does nothing on Windows.

`win.isVisibleOnAllWorkspaces()` *macOS Linux*

Returns `boolean` - Whether the window is visible on all workspaces.

Note: This API always returns false on Windows.

`win.setIgnoreMouseEvents(ignore[, options])`

- `ignore` boolean
- `options` Object (optional)
  - `forward` boolean (optional) *macOS Windows* - If true, forwards mouse move messages to Chromium, enabling mouse related events such as `mouseleave`. Only used when `ignore` is true. If `ignore` is false, forwarding is always disabled regardless of this value.

Makes the window ignore all mouse events.

All mouse events happened in this window will be passed to the window below this window, but if this window has focus, it will still receive keyboard events.

`win.setContentProtection(enable)` *macOS Windows*

- `enable` boolean

Prevents the window contents from being captured by other apps.

On macOS it sets the NSWindow's sharingType to NSWindowSharingNone. On Windows it calls SetWindowDisplayAffinity with `WDA_EXCLUDEFROMCAPTURE`. For Windows 10 version 2004 and up the window will be removed from capture entirely, older Windows versions behave as if `WDA_MONITOR` is applied capturing a black window.

`win.setFocusable(focusable)` *macOS Windows*

- `focusable` boolean

Changes whether the window can be focused.

On macOS it does not remove the focus from the window.

`win.isFocusable()` *macOS Windows*

Returns whether the window can be focused.

`win.setParentWindow(parent)`

- `parent` BrowserWindow | null

Sets `parent` as current window's parent window, passing `null` will turn current window into a top-level window.

`win.getParentWindow()`

Returns `BrowserWindow | null` - The parent window or `null` if there is no parent.

`win.getChildWindows()`

Returns `BrowserWindow[]` - All child windows.

`win.setAutoHideCursor(autoHide)` *macOS*

- `autoHide` boolean

Controls whether to hide cursor when typing.

`win.selectPreviousTab()` *macOS*

Selects the previous tab when native tabs are enabled and there are other tabs in the window.

`win.selectNextTab()` *macOS*

Selects the next tab when native tabs are enabled and there are other tabs in the window.

`win.mergeAllWindows()` *macOS*

Merges all windows into one window with multiple tabs when native tabs are enabled and there is more than one open window.

`win.moveTabToNewWindow()` *macOS*

Moves the current tab into a new window if native tabs are enabled and there is more than one tab in the current window.

`win.toggleTabBar()` *macOS*

Toggles the visibility of the tab bar if native tabs are enabled and there is only one tab in the current window.

`win.addTabbedWindow(browserWindow)` *macOS*

- `browserWindow` BrowserWindow

Adds a window as a tab on this window, after the tab for the window instance.

`win.setVibrancy(type)` *macOS*

- `type` string | null - Can be `appearance-based` , `light` , `dark` , `titlebar` , `selection` , `menu` , `popover` , `sidebar` , `medium-light` , `ultra-dark` , `header` , `sheet` , `window` , `hud` , `fullscreen-ui` , `tooltip` , `content` , `under-window` , or `under-page` . See the macOS documentation for more details.

Adds a vibrancy effect to the browser window. Passing `null` or an empty string will remove the vibrancy effect on the window.

Note that `appearance-based` , `light` , `dark` , `medium-light` , and `ultra-dark` have been deprecated and will be removed in an upcoming version of macOS.

`win.setTrafficLightPosition(position)` *macOS*

- `position` Point

Set a custom position for the traffic light buttons in frameless window.

`win.getTrafficLightPosition()` *macOS*

Returns `Point` - The custom position for the traffic light buttons in frameless window.

`win.setTouchBar(touchBar)` *macOS*

- `touchBar` TouchBar | null

Sets the touchBar layout for the current window. Specifying `null` or `undefined` clears the touch bar. This method only has an effect if the machine has a touch bar and is running on macOS 10.12.1+.

Note: The TouchBar API is currently experimental and may change or be removed in future Electron releases.

`win.setBrowserView(browserView)` *Experimental*

- `browserView` BrowserView | null - Attach `browserView` to `win` . If there are other `BrowserView` s attached, they will be removed from this window.

`win.getBrowserView()` *Experimental*

Returns `BrowserView | null` - The `BrowserView` attached to `win` . Returns `null` if one is not attached. Throws an error if multiple `BrowserView` s are attached.

`win.addBrowserView(browserView)` *Experimental*

- `browserView` BrowserView

Replacement API for setBrowserView supporting work with multi browser views.

`win.removeBrowserView(browserView)` *Experimental*

- `browserView` BrowserView

`win.setTopBrowserView(browserView)` *Experimental*

- `browserView` BrowserView

Raises `browserView` above other `BrowserView` s attached to `win` . Throws an error if `browserView` is not attached to `win` .

`win.getBrowserViews()` *Experimental*

Returns `BrowserView[]` - an array of all BrowserViews that have been attached with `addBrowserView` or `setBrowserView` .

Note: The BrowserView API is currently experimental and may change or be removed in future Electron releases.

`win.setTitleBarOverlay(options)` *Windows*

- `options` Object
  - `color` String (optional) *Windows* - The CSS color of the Window Controls Overlay when enabled.
  - `symbolColor` String (optional) *Windows* - The CSS color of the symbols on the Window Controls Overlay when enabled.
  - `height` Integer (optional) *Windows* - The height of the title bar and Window Controls Overlay in pixels.

On a Window with Window Controls Overlay already enabled, this method updates the style of the title bar overlay.