



ipcRenderer

▸ History

Communicate asynchronously from a renderer process to the main process.

Process: **Renderer**

The `ipcRenderer` module is an **EventEmitter**. It provides a few methods so you can send synchronous and asynchronous messages from the render process (web page) to the main process. You can also receive replies from the main process.

See **IPC tutorial** for code examples.

Methods

The `ipcRenderer` module has the following method to listen for events and send messages:

`ipcRenderer.on(channel, listener)`

- `channel` string
- `listener` Function
 - event **`ipcRendererEvent`** ⓘ
 - `...args` any[]

Listens to `channel`, when a new message arrives `listener` would be called with `listener(event, args...)`.

WARNING

Do not expose the `event` argument to the renderer for security reasons! Wrap any callback that you receive from the renderer in another function like this: `ipcRenderer.on('my-channel', (event, ...args) => callback(...args))`. Not wrapping the callback in such a

function would expose dangerous Electron APIs to the renderer process. See the [security guide](#) for more info.

`ipcRenderer.off(channel, listener)`

- `channel` string
- `listener` Function
 - event [ipcRendererEvent](#) ⓘ
 - `...args` any[]

Removes the specified `listener` from the listener array for the specified `channel`.

`ipcRenderer.once(channel, listener)`

- `channel` string
- `listener` Function
 - event [ipcRendererEvent](#) ⓘ
 - `...args` any[]

Adds a one time `listener` function for the event. This `listener` is invoked only the next time a message is sent to `channel`, after which it is removed.

`ipcRenderer.addListener(channel, listener)`

- `channel` string
- `listener` Function
 - event [ipcRendererEvent](#) ⓘ
 - `...args` any[]

Alias for [ipcRenderer.on](#).

`ipcRenderer.removeListener(channel, listener)`

- `channel` string
- `listener` Function

- event [IpcRendererEvent](#) ⓘ
- ...args any[]

Alias for [ipcRenderer.off](#).

`ipcRenderer.removeAllListeners([channel])`

- channel string (optional)

Removes all listeners from the specified `channel`. Removes all listeners from all channels if no channel is specified.

`ipcRenderer.send(channel, ...args)`

- channel string
- ...args any[]

Send an asynchronous message to the main process via `channel`, along with arguments. Arguments will be serialized with the [Structured Clone Algorithm](#), just like [window.postMessage](#), so prototype chains will not be included. Sending Functions, Promises, Symbols, WeakMaps, or WeakSets will throw an exception.

NOTE: Sending non-standard JavaScript types such as DOM objects or special Electron objects will throw an exception.

Since the main process does not have support for DOM objects such as `ImageBitmap`, `File`, `DOMMatrix` and so on, such objects cannot be sent over Electron's IPC to the main process, as the main process would have no way to decode them. Attempting to send such objects over IPC will result in an error.

The main process handles it by listening for `channel` with the [ipcMain](#) module.

If you need to transfer a [MessagePort](#) to the main process, use [ipcRenderer.postMessage](#).

If you want to receive a single response from the main process, like the result of a method call, consider using [ipcRenderer.invoke](#).

`ipcRenderer.invoke(channel, ...args)`

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

Returns `Promise<any>` - Resolves with the response from the main process.

Send a message to the main process via `channel` and expect a result asynchronously. Arguments will be serialized with the [Structured Clone Algorithm](#), just like `window.postMessage`, so prototype chains will not be included. Sending Functions, Promises, Symbols, WeakMaps, or WeakSets will throw an exception.

The main process should listen for `channel` with `ipcMain.handle()`.

For example:

```
// Renderer process
ipcRenderer.invoke('some-name', someArgument).then((result) => {
  // ...
})

// Main process
ipcMain.handle('some-name', async (event, someArgument) => {
  const result = await doSomeWork(someArgument)
  return result
})
```

If you need to transfer a [MessagePort](#) to the main process, use `ipcRenderer.postMessage`.

If you do not need a response to the message, consider using `ipcRenderer.send`.

NOTE

Sending non-standard JavaScript types such as DOM objects or special Electron objects will throw an exception.

Since the main process does not have support for DOM objects such as `ImageBitmap`, `File`, `DOMMatrix` and so on, such objects cannot be sent over Electron's IPC to the main process, as

the main process would have no way to decode them. Attempting to send such objects over IPC will result in an error.

NOTE

If the handler in the main process throws an error, the promise returned by `invoke` will reject. However, the `Error` object in the renderer process will not be the same as the one thrown in the main process.

`ipcRenderer.sendSync(channel, ...args)`

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

Returns `any` - The value sent back by the `ipcMain` handler.

Send a message to the main process via `channel` and expect a result synchronously. Arguments will be serialized with the **Structured Clone Algorithm**, just like `window.postMessage`, so prototype chains will not be included. Sending Functions, Promises, Symbols, WeakMaps, or WeakSets will throw an exception.

NOTE: Sending non-standard JavaScript types such as DOM objects or special Electron objects will throw an exception.

Since the main process does not have support for DOM objects such as `ImageBitmap`, `File`, `DOMMatrix` and so on, such objects cannot be sent over Electron's IPC to the main process, as the main process would have no way to decode them. Attempting to send such objects over IPC will result in an error.

The main process handles it by listening for `channel` with `ipcMain` module, and replies by setting `event.returnValue`.

WARNING

Sending a synchronous message will block the whole renderer process until the reply is received, so use this method only as a last resort. It's much better to use the asynchronous

version, [invoke\(\)](#).

`ipcRenderer.postMessage(channel, message, [transfer])`

- `channel` string
- `message` any
- `transfer` `MessagePort[]` (optional)

Send a message to the main process, optionally transferring ownership of zero or more `MessagePort` objects.

The transferred `MessagePort` objects will be available in the main process as `MessagePortMain` objects by accessing the `ports` property of the emitted event.

For example:

```
// Renderer process
const { port1, port2 } = new MessageChannel()
ipcRenderer.postMessage('port', { message: 'hello' }, [port1])


// Main process
ipcMain.on('port', (e, msg) => {
  const [port] = e.ports
  // ...
})
```

For more information on using `MessagePort` and `MessageChannel`, see the [MDN documentation](#).

`ipcRenderer.sendToHost(channel, ...args)`

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

Like `ipcRenderer.send` but the event will be sent to the `<webview>` element in the host page instead of the main process.

 [Edit this page](#)