Inter-Process Communication

Inter-Process Communication

Inter-process communication (IPC) is a key part of building feature-rich desktop applications in Electron. Because the main and renderer processes have different responsibilities in Electron's process model, IPC is the only way to perform many common tasks, such as calling a native API from your UI or triggering changes in your web contents from native menus.

IPC channels

In Electron, processes communicate by passing messages through developer-defined "channels" with the `ipcMain` and `ipcRenderer` modules. These channels are **arbitrary** (you can name them anything you want) and **bidirectional** (you can use the same channel name for both modules).

In this guide, we'll be going over some fundamental IPC patterns with concrete examples that you can use as a reference for your app code.

Understanding context-isolated processes

Before proceeding to implementation details, you should be familiar with the idea of using a preload script to import Node.js and Electron modules in a context-isolated renderer process.

- For a full overview of Electron's process model, you can read the process model docs.
- For a primer into exposing APIs from your preload script using the `contextBridge` module, check out the context isolation tutorial.

Pattern 1: Renderer to main (one-way)

To fire a one-way IPC message from a renderer process to the main process, you can use the `ipcRenderer.send` API to send a message that is then received by the `ipcMain.on` API.

You usually use this pattern to call a main process API from your web contents.

We'll demonstrate this pattern by creating a simple app that can programmatically change its window title.

For this demo, you'll need to add code to your main process, your renderer process, and a preload script. The full code is below, but we'll be explaining each file individually in the following sections.

`fiddle docs/fiddles/ipc/pattern-1`

1. Listen for events with `ipcMain.on`

In the main process, set an IPC listener on the `set-title` channel with the `ipcMain.on` API:

"'javascript {6-10,22} title='main.js (Main Process)' const {app, BrowserWindow, ipcMain} = require('electron') const path = require('path')

//...

function handleSetTitle (event, title) { const webContents = event.sender const win = BrowserWindow.fromWebContents(webContents) win.setTitle(title) }

function createWindow () { const mainWindow = new BrowserWindow({ webPreferences: { preload: path.join(___dirname, 'preload.js') } }) mainWindow.loadFile('index.html') }

app.whenReady().then(() => { ipcMain.on('set-title', handleSetTitle) createWindow() } //...

```
The above `handleSetTitle` callback has two parameters: an [IpcMainEvent] structure and a
`title` string. Whenever a message comes through the `set-title` channel, this function will
find the BrowserWindow instance attached to the message sender and use the `win.setTitle`
API on it.

:::info
Make sure you're loading the `index.html` and `preload.js` entry points for the following st
:::

### 2. Expose `ipcRenderer.send` via preload

To send messages to the listener created above, you can use the `ipcRenderer.send` API.
By default, the renderer process has no Node.js or Electron module access. As an app develop
you need to choose which APIs to expose from your preload script using the `contextBridge` A

In your preload script, add the following code, which will expose a global `window.electronA
variable to your renderer process.

```javascript title='preload.js (Preload Script)'
const { contextBridge, ipcRenderer } = require('electron')
```
```

```
contextBridge.exposeInMainWorld('electronAPI', {
    setTitle: (title) => ipcRenderer.send('set-title', title)
})
```

At this point, you'll be able to use the `window.electronAPI.setTitle()` function in the renderer process.

:::caution Security warning We don't directly expose the whole `ipcRenderer.send` API for security reasons. Make sure to limit the renderer's access to Electron APIs as much as possible. :::

3. Build the renderer process UI

In our BrowserWindow's loaded HTML file, add a basic user interface consisting of a text input and a button:

```
html {11-12} title='index.html' <!DOCTYPE html> <html>   <head>
<meta charset="UTF-8">     <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
-->     <meta http-equiv="Content-Security-Policy" content="default-src
'self'; script-src 'self'">     <title>Hello World!</title>
</head>   <body>     Title: <input id="title"/>     <button
id="btn" type="button">Set</button>     <script src="./renderer.js"></script>
</body> </html>
```

To make these elements interactive, we'll be adding a few lines of code in the imported `renderer.js` file that leverages the `window.electronAPI` functionality exposed from the preload script:

```
javascript title='renderer.js (Renderer Process)' const setButton
= document.getElementById('btn') const titleInput = document.getElementById('title')
setButton.addEventListener('click', () => {     const title =
titleInput.value     window.electronAPI.setTitle(title) });
```

At this point, your demo should be fully functional. Try using the input field and see what happens to your BrowserWindow title!

Pattern 2: Renderer to main (two-way)

A common application for two-way IPC is calling a main process module from your renderer process code and waiting for a result. This can be done by using `ipcRenderer.invoke` paired with `ipcMain.handle`.

In the following example, we'll be opening a native file dialog from the renderer process and returning the selected file's path.

For this demo, you'll need to add code to your main process, your renderer process, and a preload script. The full code is below, but we'll be explaining each file individually in the following sections.

```
fiddle docs/fiddles/ipc/pattern-2
```

1. Listen for events with `ipcMain.handle`

In the main process, we'll be creating a `handleFileOpen()` function that calls `dialog.showOpenDialog` and returns the value of the file path selected by the user. This function is used as a callback whenever an `ipcRender.invoke` message is sent through the `dialog:openFile` channel from the renderer process. The return value is then returned as a Promise to the original `invoke` call.

:::caution A word on error handling Errors thrown through `handle` in the main process are not transparent as they are serialized and only the `message` property from the original error is provided to the renderer process. Please refer to #24427 for details. :::

"'javascript {6-13,25} title='main.js (Main Process)' const { BrowserWindow, dialog, ipcMain } = require('electron') const path = require('path')

//...

async function handleFileOpen() { const { canceled, filePaths } = await dialog.showOpenDialog() if (canceled) { return } else { return filePaths[0] } }

function createWindow () { const mainWindow = new BrowserWindow({ webPreferences: { preload: path.join(__dirname, 'preload.js') } }) mainWindow.loadFile('index.html') }

app.whenReady(() => { ipcMain.handle('dialog:openFile', handleFileOpen) createWindow() }) //...

```
:::tip on channel names
The `dialog:` prefix on the IPC channel name has no effect on the code. It only serves
as a namespace that helps with code readability.
:::

:::info
Make sure you're loading the `index.html` and `preload.js` entry points for the following st
:::

### 2. Expose `ipcRenderer.invoke` via preload

In the preload script, we expose a one-line `openFile` function that calls and returns the v
`ipcRenderer.invoke('dialog:openFile')`. We'll be using this API in the next step to call th
native dialog from our renderer's user interface.

```javascript title='preload.js (Preload Script)'
const { contextBridge, ipcRenderer } = require('electron')

contextBridge.exposeInMainWorld('electronAPI', {
  openFile: () => ipcRenderer.invoke('dialog:openFile')
})
```
```

:::caution Security warning We don't directly expose the whole `ipcRenderer.invoke` API for security reasons. Make sure to limit the renderer's access to Electron APIs as much as possible. :::

3. Build the renderer process UI

Finally, let's build the HTML file that we load into our BrowserWindow.

```
html {10-11} title='index.html' <!DOCTYPE html> <html>   <head>
<meta charset="UTF-8">     <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
-->     <meta http-equiv="Content-Security-Policy" content="default-src
'self'; script-src 'self'">     <title>Dialog</title>   </head>
<body>     <button type="button" id="btn">Open a File</button>
File path: <strong id="filePath"></strong>     <script src='./renderer.js'></script>
</body> </html>
```

The UI consists of a single `#btn` button element that will be used to trigger our preload API, and a `#filePath` element that will be used to display the path of the selected file. Making these pieces work will take a few lines of code in the renderer process script:

"'javascript title='renderer.js (Renderer Process)' const btn = document.getElementById('btn') const filePathElement = document.getElementById('filePath')

btn.addEventListener('click', async () => { const filePath = await window.electronAPI.openFile() filePathElement.innerText = filePath })

```
In the above snippet, we listen for clicks on the `#btn` button, and call our
`window.electronAPI.openFile()` API to activate the native Open File dialog. We then display
selected file path in the `#filePath` element.

### Note: legacy approaches

The `ipcRenderer.invoke` API was added in Electron 7 as a developer-friendly way to tackle t
IPC from the renderer process. However, there exist a couple alternative approaches to this
pattern.

:::warning Avoid legacy approaches if possible
We recommend using `ipcRenderer.invoke` whenever possible. The following two-way renderer-to
patterns are documented for historical purposes.
:::

:::info
For the following examples, we're calling `ipcRenderer` directly from the preload script to
the code samples small.
:::

#### Using `ipcRenderer.send`
```

```
The `ipcRenderer.send` API that we used for single-way communication can also be leveraged t
perform two-way communication. This was the recommended way for asynchronous two-way communi
via IPC prior to Electron 7.

```javascript title='preload.js (Preload Script)'
// You can also put expose this code to the renderer
// process with the `contextBridge` API
const { ipcRenderer } = require('electron')

ipcRenderer.on('asynchronous-reply', (_event, arg) => {
  console.log(arg) // prints "pong" in the DevTools console
})
ipcRenderer.send('asynchronous-message', 'ping')
```

```
javascript title='main.js (Main Process)' ipcMain.on('asynchronous-message',
(event, arg) => {   console.log(arg) // prints "ping" in the Node
console   // works like `send`, but returning a message back   //
to the renderer that sent the original message   event.reply('asynchronous-reply',
'pong') })
```

There are a couple downsides to this approach:

- You need to set up a second `ipcRenderer.on` listener to handle the response in the renderer process. With `invoke`, you get the response value returned as a Promise to the original API call.
- There's no obvious way to pair the `asynchronous-reply` message to the original `asynchronous-message` one. If you have very frequent messages going back and forth through these channels, you would need to add additional app code to track each call and response individually.

Using `ipcRenderer.sendSync` The `ipcRenderer.sendSync` API sends a message to the main process and waits *synchronously* for a response.

```
javascript title='main.js (Main Process)' const { ipcMain } =
require('electron') ipcMain.on('synchronous-message', (event,
arg) => {   console.log(arg) // prints "ping" in the Node console
event.returnValue = 'pong' })
```

“javascript title='preload.js (Preload Script)' // You can also put expose this code to the renderer // process with thecontextBridge‘ API const { ipcRenderer } = require(‘electron’)

const result = ipcRenderer.sendSync(‘synchronous-message’, ‘ping’) console.log(result) // prints “pong” in the DevTools console

```
The structure of this code is very similar to the `invoke` model, but we recommend
**avoiding this API** for performance reasons. Its synchronous nature means that it'll block
renderer process until a reply is received.
```
```

Pattern 3: Main to renderer

When sending a message from the main process to a renderer process, you need to specify whic
renderer is receiving the message. Messages need to be sent to a renderer process
via its [`WebContents`] instance. This WebContents instance contains a [`send`][webcontents-
that can be used in the same way as `ipcRenderer.send`.

To demonstrate this pattern, we'll be building a number counter controlled by the native ope
system menu.

For this demo, you'll need to add code to your main process, your renderer process, and a pr
script. The full code is below, but we'll be explaining each file individually in the follow
sections.

```fiddle docs/fiddles/ipc/pattern-3
```

1. Send messages with the `webContents` module

For this demo, we'll need to first build a custom menu in the main process using Electron's `Menu` module that uses the `webContents.send` API to send an IPC message from the main process to the target renderer.

“‘javascript {11-26} title=‘main.js (Main Process)’ const {app, BrowserWindow, Menu, ipcMain} = require(‘electron’) const path = require(‘path’)

function createWindow () { const mainWindow = new BrowserWindow({ webPreferences: { preload: path.join(__dirname, ‘preload.js’) } })

const menu = Menu.buildFromTemplate([{ label: app.name, submenu: [{ click: () => mainWindow.webContents.send(‘update-counter’, 1), label: ‘Increment’, }, { click: () => mainWindow.webContents.send(‘update-counter’, -1), label: ‘Decrement’, }] }]) Menu.setApplicationMenu(menu)

mainWindow.loadFile(‘index.html’) } //...

For the purposes of the tutorial, it's important to note that the `click` handler
sends a message (either `1` or `-1`) to the renderer process through the `update-counter` ch

```javascript
click: () => mainWindow.webContents.send('update-counter', -1)
```

Make sure you're loading the `index.html` and `preload.js` entry points for the following steps!

2. Expose `ipcRenderer.on` via preload

Like in the previous renderer-to-main example, we use the `contextBridge` and `ipcRenderer` modules in the preload script to expose IPC functionality to the

renderer process:

“‘javascript title=‘preload.js (Preload Script)’ const { contextBridge, ipcRenderer } = require(‘electron’)

contextBridge.exposeInMainWorld(‘electronAPI’, { onUpdateCounter: (callback) => ipcRenderer.on(‘update-counter’, callback) })

```
After loading the preload script, your renderer process should have access to the
`window.electronAPI.onUpdateCounter()` listener function.

:::caution Security warning
We don't directly expose the whole `ipcRenderer.on` API for [security reasons]. Make sure to
limit the renderer's access to Electron APIs as much as possible.
:::

:::info
In the case of this minimal example, you can call `ipcRenderer.on` directly in the preload s
rather than exposing it over the context bridge.

```javascript title='preload.js (Preload Script)'
const { ipcRenderer } = require('electron')

window.addEventListener('DOMContentLoaded', () => {
    const counter = document.getElementById('counter')
    ipcRenderer.on('update-counter', (_event, value) => {
        const oldValue = Number(counter.innerText)
        const newValue = oldValue + value
        counter.innerText = newValue
    })
})
```

However, this approach has limited flexibility compared to exposing your preload APIs over the context bridge, since your listener can’t directly interact with your renderer code. :::

3. Build the renderer process UI

To tie it all together, we’ll create an interface in the loaded HTML file that contains a `#counter` element that we’ll use to display the values:

```
html {10} title='index.html' <!DOCTYPE html> <html>   <head>
<meta charset="UTF-8">     <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
-->     <meta http-equiv="Content-Security-Policy" content="default-src
'self'; script-src 'self'">     <title>Menu Counter</title>
</head>   <body>     Current value: <strong id="counter">0</strong>
<script src="./renderer.js"></script>   </body> </html>
```

Finally, to make the values update in the HTML document, we’ll add a few lines
```

of DOM manipulation so that the value of the `#counter` element is updated whenever we fire an `update-counter` event.

“‘javascript title=‘renderer.js (Renderer Process)’ const counter = document.getElementById(‘counter’)

window.electronAPI.onUpdateCounter((_event, value) => { const oldValue = Number(counter.innerText) const newValue = oldValue + value counter.innerText = newValue })

```
In the above code, we're passing in a callback to the `window.electronAPI.onUpdateCounter` f
exposed from our preload script. The second `value` parameter corresponds to the `1` or `-1`
were passing in from the `webContents.send` call from the native menu.

### Optional: returning a reply

There's no equivalent for `ipcRenderer.invoke` for main-to-renderer IPC. Instead, you can
send a reply back to the main process from within the `ipcRenderer.on` callback.

We can demonstrate this with slight modifications to the code from the previous example. In
renderer process, use the `event` parameter to send a reply back to the main process through
`counter-value` channel.

```javascript title='renderer.js (Renderer Process)'
const counter = document.getElementById('counter')

window.electronAPI.onUpdateCounter((event, value) => {
  const oldValue = Number(counter.innerText)
  const newValue = oldValue + value
  counter.innerText = newValue
  event.sender.send('counter-value', newValue)
})
```

In the main process, listen for `counter-value` events and handle them appropriately.

```
javascript title='main.js (Main Process)' //... ipcMain.on('counter-value',
(_event, value) => {   console.log(value) // will print value to
Node console }) //...
```

Pattern 4: Renderer to renderer

There’s no direct way to send messages between renderer processes in Electron using the `ipcMain` and `ipcRenderer` modules. To achieve this, you have two options:

- Use the main process as a message broker between renderers. This would involve sending a message from one renderer to the main process, which would forward the message to the other renderer.
```

- Pass a MessagePort from the main process to both renderers. This will allow direct communication between renderers after the initial setup.

Object serialization

Electron's IPC implementation uses the HTML standard Structured Clone Algorithm to serialize objects passed between processes, meaning that only certain types of objects can be passed through IPC channels.

In particular, DOM objects (e.g. `Element`, `Location` and `DOMMatrix`), Node.js objects backed by C++ classes (e.g. `process.env`, some members of `Stream`), and Electron objects backed by C++ classes (e.g. `WebContents`, `BrowserWindow` and `WebFrame`) are not serializable with Structured Clone.