Process Model

Electron inherits its multi-process architecture from Chromium, which makes the framework architecturally very similar to a modern web browser. In this guide, we'll expound on the conceptual knowledge of Electron that we applied in the minimal quick start app.

Why not a single process?

Web browsers are incredibly complicated applications. Aside from their primary ability to display web content, they have many secondary responsibilities, such as managing multiple windows (or tabs) and loading third-party extensions.

In the earlier days, browsers usually used a single process for all of this functionality. Although this pattern meant less overhead for each tab you had open, it also meant that one website crashing or hanging would affect the entire browser.

The multi-process model

To solve this problem, the Chrome team decided that each tab would render in its own process, limiting the harm that buggy or malicious code on a web page could cause to the app as a whole. A single browser process then controls these processes, as well as the application lifecycle as a whole. This diagram below from the Chrome Comic visualizes this model:


Chrome's multi-process architecture

Electron applications are structured very similarly. As an app developer, you control two types of processes: main and renderer. These are analogous to Chrome's own browser and renderer processes outlined above.

The main process

Each Electron app has a single main process, which acts as the application's entry point. The main process runs in a Node.js environment, meaning it has the ability to `require` modules and use all of Node.js APIs.

Window management

The primary purpose of the main process is to create and manage application windows with the `BrowserWindow` module.

Each instance of the `BrowserWindow` class creates an application window that loads a web page in a separate renderer process. You can interact with this web content from the main process using the window's `webContents` object.

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

const win = new BrowserWindow({ width: 800, height: 1500 })
win.loadURL('https://github.com')

const contents = win.webContents
console.log(contents)
```

> *Note: A renderer process is also created for web embeds such as the `BrowserView` module. The `webContents` object is also accessible for embedded web content.*

Because the `BrowserWindow` module is an `EventEmitter`, you can also add handlers for various user events (for example, minimizing or maximizing your window).

When a `BrowserWindow` instance is destroyed, its corresponding renderer process gets terminated as well.

Application lifecycle

The main process also controls your application's lifecycle through Electron's `app` module. This module provides a large set of events and methods that you can use to add custom application behaviour (for instance, programatically quitting your application, modifying the application dock, or showing an About panel).

As a practical example, the app shown in the quick start guide uses `app` APIs to create a more native application window experience.

```
// quitting the app when no windows are open on non-macOS platforms
app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') app.quit()
})
```

Native APIs

To extend Electron's features beyond being a Chromium wrapper for web contents, the main process also adds custom APIs to interact with the user's operating system. Electron exposes various modules that control native desktop functionality, such as menus, dialogs, and tray icons.

For a full list of Electron's main process modules, check out our API documentation.

The renderer process

Each Electron app spawns a separate renderer process for each open `BrowserWindow` (and each web embed). As its name implies, a renderer is responsible for *rendering* web content. For all intents and purposes, code ran in renderer processes should behave according to web standards (insofar as Chromium does, at least).

Therefore, all user interfaces and app functionality within a single browser window should be written with the same tools and paradigms that you use on the web.

Although explaining every web spec is out of scope for this guide, the bare minimum to understand is:

- An HTML file is your entry point for the renderer process.
- UI styling is added through Cascading Style Sheets (CSS).
- Executable JavaScript code can be added through `<script>` elements.

Moreover, this also means that the renderer has no direct access to `require` or other Node.js APIs. In order to directly include NPM modules in the renderer, you must use the same bundler toolchains (for example, `webpack` or `parcel`) that you use on the web.

> *Note: Renderer processes can be spawned with a full Node.js environment for ease of development. Historically, this used to be the default, but this feature was disabled for security reasons.*

At this point, you might be wondering how your renderer process user interfaces can interact with Node.js and Electron's native desktop functionality if these features are only accessible from the main process. In fact, there is no

direct way to import Electron's content scripts.

Preload scripts

Preload scripts contain code that executes in a renderer process before its web content begins loading. These scripts run within the renderer context, but are granted more privileges by having access to Node.js APIs.

A preload script can be attached to the main process in the `BrowserWindow` constructor's `webPreferences` option.

```
const { BrowserWindow } = require('electron')
//...
const win = new BrowserWindow({
  webPreferences: {
    preload: 'path/to/preload.js'
  }
})
//...
```

Because the preload script shares a global `Window` interface with the renderers and can access Node.js APIs, it serves to enhance your renderer by exposing arbitrary APIs in the `window` global that your web contents can then consume.

Although preload scripts share a `window` global with the renderer they're attached to, you cannot directly attach any variables from the preload script to `window` because of the `contextIsolation` default.

```
window.myAPI = {
  desktop: true
}
```

```
console.log(window.myAPI)
// => undefined
```

Context Isolation means that preload scripts are isolated from the renderer's main world to avoid leaking any privileged APIs into your web content's code.

Instead, use the `contextBridge` module to accomplish this securely:

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

contextBridge.exposeInMainWorld('myAPI', {
  desktop: true
})
```

```
console.log(window.myAPI)
// => { desktop: true }
```

This feature is incredibly useful for two main purposes:

- By exposing `ipcRenderer` helpers to the renderer, you can use inter-process communication (IPC) to trigger main process tasks from the renderer (and vice-versa).
- If you're developing an Electron wrapper for an existing web app hosted on a remote URL, you can add custom properties onto the renderer's `window` global that can be used for desktop-only logic on the web client's side.