Electron Documentation

Electron Application Architecture

Before we can dive into Electron's APIs, we need to discuss the two process types available in Electron. They are fundamentally different and important to understand.

Main and Renderer Processes

In Electron, the process that runs `package.json`'s `main` script is called **the main process**. The script that runs in the main process can display a GUI by creating web pages. An Electron app always has one main process, but never more.

Since Electron uses Chromium for displaying web pages, Chromium's multi-process architecture is also used. Each web page in Electron runs in its own process, which is called **the renderer process**.

In normal browsers, web pages usually run in a sandboxed environment and are not allowed access to native resources. Electron users, however, have the power to use Node.js APIs in web pages allowing lower level operating system interactions.

Differences Between Main Process and Renderer Process

The main process creates web pages by creating `BrowserWindow` instances. Each `BrowserWindow` instance runs the web page in its own renderer process. When a `BrowserWindow` instance is destroyed, the corresponding renderer process is also terminated.

The main process manages all web pages and their corresponding renderer processes. Each renderer process is isolated and only cares about the web page running in it.

In web pages, calling native GUI related APIs is not allowed because managing native GUI resources in web pages is very dangerous and it is easy to leak resources. If you want to perform GUI operations in a web page, the renderer process of the web page must communicate with the main process to request that the main process perform those operations.

Aside: Communication Between Processes

In Electron, we have several ways to communicate between the main process and renderer processes, such as `ipcRenderer` and `ipcMain` modules for sending messages, and the remote module for RPC style communication. There is also an FAQ entry on how to share data between web pages.

Using Electron APIs

Electron offers a number of APIs that support the development of a desktop application in both the main process and the renderer process. In both processes, you'd access Electron's APIs by requiring its included module:

```
const electron = require('electron')
```

All Electron APIs are assigned a process type. Many of them can only be used from the main process, some of them only from a renderer process, some from both. The documentation for each individual API will state which process it can be used from.

A window in Electron is for instance created using the `BrowserWindow` class. It is only available in the main process.

```
// This will work in the main process, but be `undefined` in a
// renderer process:
const { BrowserWindow } = require('electron')

const win = new BrowserWindow()
```

Since communication between the processes is possible, a renderer process can call upon the main process to perform tasks. Electron comes with a module called `remote` that exposes APIs usually only available on the main process. In order to create a `BrowserWindow` from a renderer process, we'd use the remote as a middle-man:

```
// This will work in a renderer process, but be `undefined` in the
// main process:
const { remote } = require('electron')
const { BrowserWindow } = remote

const win = new BrowserWindow()
```

Using Node.js APIs

Electron exposes full access to Node.js both in the main and the renderer process. This has two important implications:

1) All APIs available in Node.js are available in Electron. Calling the following code from an Electron app works:

```
const fs = require('fs')

const root = fs.readdirSync('/')

// This will print all files at the root-level of the disk,
// either '/' or 'C:\'.
console.log(root)
```

As you might already be able to guess, this has important security implications if you ever attempt to load remote content. You can find more information and guidance on loading remote content in our security documentation.

2) You can use Node.js modules in your application. Pick your favorite npm module. npm offers currently the world's biggest repository of open-source code – the ability to use well-maintained and tested code that used to be reserved for server applications is one of the key features of Electron.

As an example, to use the official AWS SDK in your application, you'd first install it as a dependency:

```
npm install --save aws-sdk
```

Then, in your Electron app, require and use the module as if you were building a Node.js application:

```
// A ready-to-use S3 Client
const S3 = require('aws-sdk/clients/s3')
```

There is one important caveat: Native Node.js modules (that is, modules that require compilation of native code before they can be used) will need to be compiled to be used with Electron.

The vast majority of Node.js modules are *not* native. Only 400 out of the ~650,000 modules are native. However, if you do need native modules, please consult this guide on how to recompile them for Electron.