Electron FAQ

Why am I having trouble installing Electron?

When running `npm install electron`, some users occasionally encounter installation errors.

In almost all cases, these errors are the result of network problems and not actual issues with the `electron` npm package. Errors like `ELIFECYCLE`, `EAI_AGAIN`, `ECONNRESET`, and `ETIMEDOUT` are all indications of such network problems. The best resolution is to try switching networks, or wait a bit and try installing again.

You can also attempt to download Electron directly from electron/electron/releases if installing via `npm` is failing.

When will Electron upgrade to latest Chrome?

The Chrome version of Electron is usually bumped within one or two weeks after a new stable Chrome version gets released. This estimate is not guaranteed and depends on the amount of work involved with upgrading.

Only the stable channel of Chrome is used. If an important fix is in beta or dev channel, we will back-port it.

For more information, please see the security introduction.

When will Electron upgrade to latest Node.js?

When a new version of Node.js gets released, we usually wait for about a month before upgrading the one in Electron. So we can avoid getting affected by bugs introduced in new Node.js versions, which happens very often.

New features of Node.js are usually brought by V8 upgrades, since Electron is using the V8 shipped by Chrome browser, the shiny new JavaScript feature of a new Node.js version is usually already in Electron.

How to share data between web pages?

To share data between web pages (the renderer processes) the simplest way is to use HTML5 APIs which are already available in browsers. Good candidates are Storage API, `localStorage`, `sessionStorage`, and IndexedDB.

Alternatively, you can use the IPC primitives that are provided by Electron. To share data between the main and renderer processes, you can use the `ipcMain` and `ipcRenderer` modules. To communicate directly between web pages, you can send a `MessagePort` from one to the other, possibly via the main process using `ipcRenderer.postMessage()`. Subsequent communication over message ports is direct and does not detour through the main process.

My app's tray disappeared after a few minutes.

This happens when the variable which is used to store the tray gets garbage collected.

If you encounter this problem, the following articles may prove helpful:

- Memory Management
- Variable Scope

If you want a quick fix, you can make the variables global by changing your code from this:

```
const { app, Tray } = require('electron')
app.whenReady().then(() => {
  const tray = new Tray('/path/to/icon.png')
  tray.setTitle('hello world')
})
```

to this:

```
const { app, Tray } = require('electron')
let tray = null
app.whenReady().then(() => {
  tray = new Tray('/path/to/icon.png')
  tray.setTitle('hello world')
})
```

I can not use jQuery/RequireJS/Meteor/AngularJS in Electron.

Due to the Node.js integration of Electron, there are some extra symbols inserted into the DOM like `module`, `exports`, `require`. This causes problems for some libraries since they want to insert the symbols with the same names.

To solve this, you can turn off node integration in Electron:

```
// In the main process.
const { BrowserWindow } = require('electron')
const win = new BrowserWindow({
  webPreferences: {
    nodeIntegration: false
  }
})
win.show()
```

But if you want to keep the abilities of using Node.js and Electron APIs, you have to rename the symbols in the page before including other libraries:

```
<head>
<script>
```

```
window.nodeRequire = require;
delete window.require;
delete window.exports;
delete window.module;
</script>
<script type="text/javascript" src="jquery.js"></script>
</head>
```

`require('electron').xxx` is undefined.

When using Electron's built-in module you might encounter an error like this:

```
> require('electron').webFrame.setZoomFactor(1.0)
Uncaught TypeError: Cannot read property 'setZoomLevel' of undefined
```

It is very likely you are using the module in the wrong process. For example `electron.app` can only be used in the main process, while `electron.webFrame` is only available in renderer processes.

The font looks blurry, what is this and what can I do?

If sub-pixel anti-aliasing is deactivated, then fonts on LCD screens can look blurry. Example:

subpixel rendering example

Sub-pixel anti-aliasing needs a non-transparent background of the layer containing the font glyphs. (See this issue for more info).

To achieve this goal, set the background in the constructor for BrowserWindow:

```
const { BrowserWindow } = require('electron')
const win = new BrowserWindow({
  backgroundColor: '#fff'
})
```

The effect is visible only on (some?) LCD screens. Even if you don't see a difference, some of your users may. It is best to always set the background this way, unless you have reasons not to do so.

Notice that just setting the background in the CSS does not have the desired effect.