# Glossary

This page defines some terminology that is commonly used in Electron development.

### ASAR

ASAR stands for Atom Shell Archive Format. An asar archive is a simple `tar`-like format that concatenates files into a single file. Electron can read arbitrary files from it without unpacking the whole file.

The ASAR format was created primarily to improve performance on Windows when reading large quantities of small files (e.g. when loading your app's JavaScript dependency tree from `node_modules`).

### code signing

Code signing is a process where an app developer digitally signs their code to ensure that it hasn't been tampered with after packaging. Both Windows and macOS implement their own version of code signing. As a desktop app developer, it's important that you sign your code if you plan on distributing it to the general public.

For more information, read the Code Signing tutorial.

### context isolation

Context isolation is a security measure in Electron that ensures that your preload script cannot leak privileged Electron or Node.js APIs to the web contents in your renderer process. With context isolation enabled, the only way to expose APIs from your preload script is through the `contextBridge` API.

For more information, read the Context Isolation tutorial.

See also: preload script, renderer process

### CRT

The C Runtime Library (CRT) is the part of the C++ Standard Library that incorporates the ISO C99 standard library. The Visual C++ libraries that implement the CRT support native code development, and both mixed native and managed code, and pure managed code for .NET development.

### DMG

An Apple Disk Image is a packaging format used by macOS. DMG files are commonly used for distributing application "installers".

### IME

Input Method Editor. A program that allows users to enter characters and symbols not found on their keyboard. For example, this allows users of Latin keyboards to input Chinese, Japanese, Korean and Indic characters.

### IDL

Interface description language. Write function signatures and data types in a format that can be used to generate interfaces in Java, C++, JavaScript, etc.

### IPC

IPC stands for inter-process communication. Electron uses IPC to send serialized JSON messages between the main and renderer processes.

see also: main process, renderer process

### main process

The main process, commonly a file named `main.js`, is the entry point to every Electron app. It controls the life of the app, from open to close. It also manages native elements such as the Menu, Menu Bar, Dock, Tray, etc. The main process is responsible for creating each new renderer process in the app. The full Node API is built in.

Every app's main process file is specified in the `main` property in `package.json`. This is how `electron .` knows what file to execute at startup.

In Chromium, this process is referred to as the "browser process". It is renamed in Electron to avoid confusion with renderer processes.

See also: process, renderer process

### MAS

Acronym for Apple's Mac App Store. For details on submitting your app to the MAS, see the Mac App Store Submission Guide.

### Mojo

An IPC system for communicating intra- or inter-process, and that's important because Chrome is keen on being able to split its work into separate processes or not, depending on memory pressures etc.

See https://chromium.googlesource.com/chromium/src/+/main/mojo/README.md

See also: IPC

### MSI

On Windows, MSI packages are used by the Windows Installer (also known as Microsoft Installer) service to install and configure applications.

More information can be found in Microsoft's documentation.

### native modules

Native modules (also called addons in Node.js) are modules written in C or C++ that can be loaded into Node.js or Electron using the require() function, and used as if they were an ordinary Node.js module. They are used primarily to provide an interface between JavaScript running in Node.js and C/C++ libraries.

Native Node modules are supported by Electron, but since Electron is very likely to use a different V8 version from the Node binary installed in your system, you have to manually specify the location of Electron's headers when building native modules.

For more information, read the [Native Node Modules] tutorial.

### notarization

Notarization is a macOS-specific process where a developer can send a code-signed app to Apple servers to get verified for malicious components through an automated service.

See also: code signing

### OSR

OSR (offscreen rendering) can be used for loading heavy page in background and then displaying it after (it will be much faster). It allows you to render page without showing it on screen.

For more information, read the [Offscreen Rendering][osr] tutorial.

### preload script

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

See also: renderer process, context isolation

### process

A process is an instance of a computer program that is being executed. Electron apps that make use of the main and one or many renderer process are actually running several programs simultaneously.

In Node.js and Electron, each running process has a `process` object. This object is a global that provides information about, and control over, the current process. As a global, it is always available to applications without using require().

See also: main process, renderer process

### renderer process

The renderer process is a browser window in your app. Unlike the main process, there can be multiple of these and each is run in a separate process. They can also be hidden.

See also: process, main process

### sandbox

The sandbox is a security feature inherited from Chromium that restricts your renderer processes to a limited set of permissions.

For more information, read the Process Sandboxing tutorial.

See also: process

### Squirrel

Squirrel is an open-source framework that enables Electron apps to update automatically as new versions are released. See the autoUpdater API for info about getting started with Squirrel.

### userland

This term originated in the Unix community, where "userland" or "userspace" referred to programs that run outside of the operating system kernel. More recently, the term has been popularized in the Node and npm community to distinguish between the features available in "Node core" versus packages published to the npm registry by the much larger "user" community.

Like Node, Electron is focused on having a small set of APIs that provide all the necessary primitives for developing multi-platform desktop applications. This design philosophy allows Electron to remain a flexible tool without being

overly prescriptive about how it should be used. Userland enables users to create and share tools that provide additional functionality on top of what is available in "core".

### V8

V8 is Google's open source JavaScript engine. It is written in C++ and is used in Google Chrome. V8 can run standalone, or can be embedded into any C++ application.

Electron builds V8 as part of Chromium and then points Node to that V8 when building it.

V8's version numbers always correspond to those of Google Chrome. Chrome 59 includes V8 5.9, Chrome 58 includes V8 5.8, etc.

- v8.dev
- nodejs.org/api/v8.html
- docs/development/v8-development.md

### webview

`webview` tags are used to embed 'guest' content (such as external web pages) in your Electron app. They are similar to `iframe`s, but differ in that each webview runs in a separate process. It doesn't have the same permissions as your web page and all interactions between your app and embedded content will be asynchronous. This keeps your app safe from the embedded content.