Table of Contents

Please make sure that you use the documents that match your Electron version. The version number should be a part of the page URL. If it's not, you are probably using the documentation of a development branch which may contain API changes that are not compatible with your Electron version. To view older versions of the documentation, you can browse by tag on GitHub by opening the "Switch branches/tags" dropdown and selecting the tag that matches your version.

FAQ

There are questions that are asked quite often. Check this out before creating an issue:

- Electron FAQ

Guides

- Glossary of Terms
- Supported Platforms
- Security
- Electron Versioning
- Application Distribution
- Mac App Store Submission Guide
- Windows Store Guide
- Application Packaging
- Using Native Node Modules
- Debugging Main Process
- Using Selenium and WebDriver
- DevTools Extension
- Using Pepper Flash Plugin
- Using Widevine CDM Plugin
- Testing on Headless CI Systems (Travis, Jenkins)
- Offscreen Rendering
- Keyboard Shortcuts

Tutorials

- Quick Start
- Desktop Environment Integration
- Online/Offline Event Detection
- REPL

API References

- Synopsis
- Process Object
- Supported Chrome Command Line Switches
- Environment Variables

Custom DOM Elements:

- `File` Object
- `<webview>` Tag
- `window.open` Function

Modules for the Main Process:

Modules for the Renderer Process (Web Page):

Modules for Both Processes:

Development

Electron FAQ

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.

Or you can use the IPC system, which is specific to Electron, to store objects in the main process as a global variable, and then to access them from the renderers through the `remote` property of `electron` module:

```
// In the main process.
global.sharedObject = {
  someProperty: 'default value'
}
```

```
// In page 1.
require('electron').remote.getGlobal('sharedObject').someProperty = 'new value'
```

```
// In page 2.
console.log(require('electron').remote.getGlobal('sharedObject').someProperty)
```

My app's window/tray disappeared after a few minutes.

This happens when the variable which is used to store the window/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.on('ready', () => {
  const tray = new Tray('/path/to/icon.png')
  tray.setTitle('hello world')
})
```

to this:

```
const {app, Tray} = require('electron')
let tray = null
app.on('ready', () => {
  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')
let 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
```

This is because you have the npm `electron` module installed either locally or globally, which overrides Electron's built-in module.

To verify whether you are using the correct built-in module, you can print the path of the `electron` module:

```
console.log(require.resolve('electron'))
```

and then check if it is in the following form:

```
"/path/to/Electron.app/Contents/Resources/atom.asar/renderer/api/lib/exports/electron.js"
```

If it is something like `node_modules/electron/index.js`, then you have to either remove the npm `electron` module, or rename it.

```
npm uninstall electron
npm uninstall -g electron
```

However if your are using the built-in module but still getting this error, 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.

Supported Platforms

Following platforms are supported by Electron:

macOS

Only 64bit binaries are provided for macOS, and the minimum macOS version supported is macOS 10.9.

Windows

Windows 7 and later are supported, older operating systems are not supported (and do not work).

Both `ia32` (`x86`) and `x64` (`amd64`) binaries are provided for Windows. Please note, the `ARM` version of Windows is not supported for now.

Linux

The prebuilt `ia32` (`i686`) and `x64` (`amd64`) binaries of Electron are built on Ubuntu 12.04, the `arm` binary is built against ARM v7 with hard-float ABI and NEON for Debian Wheezy.

Whether the prebuilt binary can run on a distribution depends on whether the distribution includes the libraries that Electron is linked to on the building platform, so only Ubuntu 12.04 is guaranteed to work, but following platforms are also verified to be able to run the prebuilt binaries of Electron:

- Ubuntu 12.04 and later
- Fedora 21
- Debian 8

Security, Native Capabilities, and Your Responsibility

As web developers, we usually enjoy the strong security net of the browser - the risks associated with the code we write are relatively small. Our websites are granted limited powers in a sandbox, and we trust that our users enjoy a browser built by a large team of engineers that is able to quickly respond to newly discovered security threats.

When working with Electron, it is important to understand that Electron is not a web browser. It allows you to build feature-rich desktop applications with familiar web technologies, but your code wields much greater power. JavaScript can access the filesystem, user shell, and more. This allows you to build high quality native applications, but the inherent security risks scale with the additional powers granted to your code.

With that in mind, be aware that displaying arbitrary content from untrusted sources poses a severe security risk that Electron is not intended to handle. In fact, the most popular Electron apps (Atom, Slack, Visual Studio Code, etc) display primarily local content (or trusted, secure remote content without Node integration) – if your application executes code from an online source, it is your responsibility to ensure that the code is not malicious.

Reporting Security Issues

For information on how to properly disclose an Electron vulnerability, see SECURITY.md

Chromium Security Issues and Upgrades

While Electron strives to support new versions of Chromium as soon as possible, developers should be aware that upgrading is a serious undertaking - involving hand-editing dozens or even hundreds of files. Given the resources and contributions available today, Electron will often not be on the very latest version of Chromium, lagging behind by either days or weeks.

We feel that our current system of updating the Chromium component strikes an appropriate balance between the resources we have available and the needs of the majority of applications built on top of the framework. We definitely are interested in hearing more about specific use cases from the people that build things on top of Electron. Pull requests and contributions supporting this effort are always very welcome.

Ignoring Above Advice

A security issue exists whenever you receive code from a remote destination and execute it locally. As an example, consider a remote website being displayed inside a browser window. If an attacker somehow manages to change said content (either by attacking the source directly, or by sitting between your app and the actual destination), they will be able to execute native code on the user's machine.

> :warning: Under no circumstances should you load and execute remote code with Node integration enabled. Instead, use only local files (packaged together with your application) to execute Node code. To display remote content, use the `webview` tag and make sure to disable the `nodeIntegration`.

Checklist

This is not bulletproof, but at the least, you should attempt the following:

- Only display secure (https) content
- Disable the Node integration in all renderers that display remote content (setting `nodeIntegration` to `false` in `webPreferences`)
- Enable context isolation in all renderers that display remote content (setting `contextIsolation` to `true` in `webPreferences`)

- Use `ses.setPermissionRequestHandler()` in all sessions that load remote content
- Do not disable `webSecurity` . Disabling it will disable the same-origin policy.
- Define a `Content-Security-Policy` , and use restrictive rules (i.e. `script-src 'self'`)
- Override and disable `eval` , which allows strings to be executed as code.
- Do not set `allowRunningInsecureContent` to true.
- Do not enable `experimentalFeatures` or `experimentalCanvasFeatures` unless you know what you're doing.
- Do not use `blinkFeatures` unless you know what you're doing.
- WebViews: Do not add the `nodeintegration` attribute.
- WebViews: Do not use `disablewebsecurity`
- WebViews: Do not use `allowpopups`
- WebViews: Do not use `insertCSS` or `executeJavaScript` with remote CSS/JS.

Again, this list merely minimizes the risk, it does not remove it. If your goal is to display a website, a browser will be a more secure option.

Electron Versioning

If you are a seasoned Node developer, you are surely aware of `semver` - and might be used to giving your dependency management systems only rough guidelines rather than fixed version numbers. Due to the hard dependency on Node and Chromium, Electron is in a slightly more difficult position and does not follow semver. You should therefore always reference a specific version of Electron.

Version numbers are bumped using the following rules:

- Major: For breaking changes in Electron's API - if you upgrade from `0.37.0` to `1.0.0`, you will have to update your app.
- Minor: For major Chrome and minor Node upgrades; or significant Electron changes - if you upgrade from `1.0.0` to `1.1.0`, your app is supposed to still work, but you might have to work around small changes.
- Patch: For new features and bug fixes - if you upgrade from `1.0.0` to `1.0.1`, your app will continue to work as-is.

If you are using `electron` or `electron-prebuilt`, we recommend that you set a fixed version number (`1.1.0` instead of `^1.1.0`) to ensure that all upgrades of Electron are a manual operation made by you, the developer.

Application Distribution

To distribute your app with Electron, you need to download Electron's prebuilt binaries. Next, the folder containing your app should be named `app` and placed in Electron's resources directory as shown in the following examples. Note that the location of Electron's prebuilt binaries is indicated with `electron/` in the examples below.

On macOS:

```
electron/Electron.app/Contents/Resources/app/
├── package.json
├── main.js
└── index.html
```

On Windows and Linux:

```
electron/resources/app
├── package.json
├── main.js
└── index.html
```

Then execute `Electron.app` (or `electron` on Linux, `electron.exe` on Windows), and Electron will start as your app. The `electron` directory will then be your distribution to deliver to final users.

Packaging Your App into a File

Apart from shipping your app by copying all of its source files, you can also package your app into an asar archive to avoid exposing your app's source code to users.

To use an `asar` archive to replace the `app` folder, you need to rename the archive to `app.asar`, and put it under Electron's resources directory like below, and Electron will then try to read the archive and start from it.

On macOS:

```
electron/Electron.app/Contents/Resources/
└── app.asar
```

On Windows and Linux:

```
electron/resources/
└── app.asar
```

More details can be found in Application packaging.

Rebranding with Downloaded Binaries

After bundling your app into Electron, you will want to rebrand Electron before distributing it to users.

Windows

You can rename `electron.exe` to any name you like, and edit its icon and other information with tools like rcedit.

macOS

You can rename `Electron.app` to any name you want, and you also have to rename the `CFBundleDisplayName` , `CFBundleIdentifier` and `CFBundleName` fields in the following files:

- `Electron.app/Contents/Info.plist`
- `Electron.app/Contents/Frameworks/Electron Helper.app/Contents/Info.plist`

You can also rename the helper app to avoid showing `Electron Helper` in the Activity Monitor, but make sure you have renamed the helper app's executable file's name.

The structure of a renamed app would be like:

```
MyApp.app/Contents
├── Info.plist
├── MacOS/
│   └── MyApp
└── Frameworks/
    ├── MyApp Helper EH.app
    |   ├── Info.plist
    |   └── MacOS/
    |       └── MyApp Helper EH
    ├── MyApp Helper NP.app
    |   ├── Info.plist
    |   └── MacOS/
    |       └── MyApp Helper NP
    └── MyApp Helper.app
        ├── Info.plist
        └── MacOS/
            └── MyApp Helper
```

Linux

You can rename the `electron` executable to any name you like.

Packaging Tools

Apart from packaging your app manually, you can also choose to use third party packaging tools to do the work for you:

- electron-builder
- electron-packager

Rebranding by Rebuilding Electron from Source

It is also possible to rebrand Electron by changing the product name and building it from source. To do this you need to modify the `atom.gyp` file and have a clean rebuild.

grunt-build-atom-shell

Manually checking out Electron's code and rebuilding could be complicated, so a Grunt task has been created that will handle this automatically: grunt-build-atom-shell.

This task will automatically handle editing the `.gyp` file, building from source, then rebuilding your app's native Node modules to match the new executable name.

Creating a Custom Electron Fork

Creating a custom fork of Electron is almost certainly not something you will need to do in order to build your app, even for "Production Level" applications. Using a tool such as `electron-packager` or `electron-builder` will allow you to "Rebrand" Electron without having to do these steps.

You need to fork Electron when you have custom C++ code that you have patched directly into Electron, that either cannot be upstreamed, or has been rejected from the official version. As maintainers of Electron, we very much would like to make your scenario work, so please try as hard as you can to get your changes into the official version of Electron, it will be much much easier on you, and we appreciate your help.

Creating a Custom Release with surf-build

1. Install Surf, via npm: `npm install -g surf-build@latest`
2. Create a new S3 bucket and create the following empty directory structure:

   ```
   - atom-shell/
     - symbols/
     - dist/
   ```

3. Set the following Environment Variables:
   - `ELECTRON_GITHUB_TOKEN` - a token that can create releases on GitHub
   - `ELECTRON_S3_ACCESS_KEY` , `ELECTRON_S3_BUCKET` , `ELECTRON_S3_SECRET_KEY` - the place where you'll upload node.js headers as well as symbols
   - `ELECTRON_RELEASE` - Set to `true` and the upload part will run, leave unset and `surf-build` will just do CI-type checks, appropriate to run for every pull request.
   - `CI` - Set to `true` or else it will fail
   - `GITHUB_TOKEN` - set it to the same as `ELECTRON_GITHUB_TOKEN`
   - `SURF_TEMP` - set to `C:\Temp` on Windows to prevent path too long issues
   - `TARGET_ARCH` - set to `ia32` or `x64`
4. In `script/upload.py` , you *must* set `ELECTRON_REPO` to your fork (`MYORG/electron`), especially if you are a contributor to Electron proper.
5. `surf-build -r https://github.com/MYORG/electron -s YOUR_COMMIT -n 'surf-PLATFORM-ARCH'`
6. Wait a very, very long time for the build to complete.

Mac App Store Submission Guide

Since v0.34.0, Electron allows submitting packaged apps to the Mac App Store (MAS). This guide provides information on: how to submit your app and the limitations of the MAS build.

Note: Submitting an app to Mac App Store requires enrolling Apple Developer Program, which costs money.

How to Submit Your App

The following steps introduce a simple way to submit your app to Mac App Store. However, these steps do not ensure your app will be approved by Apple; you still need to read Apple's Submitting Your App guide on how to meet the Mac App Store requirements.

Get Certificate

To submit your app to the Mac App Store, you first must get a certificate from Apple. You can follow these existing guides on web.

Get Team ID

Before signing your app, you need to know the Team ID of your account. To locate your Team ID, Sign in to Apple Developer Center, and click Membership in the sidebar. Your Team ID appears in the Membership Information section under the team name.

Sign Your App

After finishing the preparation work, you can package your app by following Application Distribution, and then proceed to signing your app.

First, you have to add a `ElectronTeamID` key to your app's `Info.plist`, which has your Team ID as value:

```
<plist version="1.0">
<dict>
  ...
  <key>ElectronTeamID</key>
  <string>TEAM_ID</string>
</dict>
</plist>
```

Then, you need to prepare two entitlements files.

`child.plist`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>com.apple.security.app-sandbox</key>
    <true/>
    <key>com.apple.security.inherit</key>
    <true/>
  </dict>
</plist>
```

`parent.plist`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>com.apple.security.app-sandbox</key>
    <true/>
    <key>com.apple.security.application-groups</key>
    <string>TEAM_ID.your.bundle.id</string>
  </dict>
</plist>
```

You have to replace `TEAM_ID` with your Team ID, and replace `your.bundle.id` with the Bundle ID of your app.

And then sign your app with the following script:

```
#!/bin/bash

# Name of your app.
APP="YourApp"
# The path of your app to sign.
APP_PATH="/path/to/YourApp.app"
# The path to the location you want to put the signed package.
RESULT_PATH="~/Desktop/$APP.pkg"
# The name of certificates you requested.
APP_KEY="3rd Party Mac Developer Application: Company Name (APPIDENTITY)"
INSTALLER_KEY="3rd Party Mac Developer Installer: Company Name (APPIDENTITY)"
# The path of your plist files.
CHILD_PLIST="/path/to/child.plist"
PARENT_PLIST="/path/to/parent.plist"

FRAMEWORKS_PATH="$APP_PATH/Contents/Frameworks"

codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/Electron Framework.framework/Versions/A/Ele
ctron Framework"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/Electron Framework.framework/Versions/A/Lib
raries/libffmpeg.dylib"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/Electron Framework.framework/Versions/A/Lib
raries/libnode.dylib"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/Electron Framework.framework"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/$APP Helper.app/Contents/MacOS/$APP Helper"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/$APP Helper.app/"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/$APP Helper EH.app/Contents/MacOS/$APP Help
er EH"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/$APP Helper EH.app/"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/$APP Helper NP.app/Contents/MacOS/$APP Help
er NP"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$FRAMEWORKS_PATH/$APP Helper NP.app/"
codesign -s "$APP_KEY" -f --entitlements "$CHILD_PLIST" "$APP_PATH/Contents/MacOS/$APP"
codesign -s "$APP_KEY" -f --entitlements "$PARENT_PLIST" "$APP_PATH"

productbuild --component "$APP_PATH" /Applications --sign "$INSTALLER_KEY" "$RESULT_PATH"
```

If you are new to app sandboxing under macOS, you should also read through Apple's Enabling App Sandbox to have a basic idea, then add keys for the permissions needed by your app to the entitlements files.

Apart from manually signing your app, you can also choose to use the electron-osx-sign module to do the job.

Sign Native Modules

Native modules used in your app also need to be signed. If using electron-osx-sign, be sure to include the path to the built binaries in the argument list:

```
electron-osx-sign YourApp.app YourApp.app/Contents/Resources/app/node_modules/nativemodule/build/release/nativemodule
```

Also note that native modules may have intermediate files produced which should not be included (as they would also need to be signed). If you use electron-packager before version 8.1.0, add `--ignore=.+\.o$` to your build step to ignore these files. Versions 8.1.0 and later ignores those files by default.

Upload Your App

After signing your app, you can use Application Loader to upload it to iTunes Connect for processing, making sure you have created a record before uploading.

Submit Your App for Review

After these steps, you can submit your app for review.

Limitations of MAS Build

In order to satisfy all requirements for app sandboxing, the following modules have been disabled in the MAS build:

- `crashReporter`
- `autoUpdater`

and the following behaviors have been changed:

- Video capture may not work for some machines.
- Certain accessibility features may not work.
- Apps will not be aware of DNS changes.
- APIs for launching apps at login are disabled. See https://github.com/electron/electron/issues/7312#issuecomment-249479237

Also, due to the usage of app sandboxing, the resources which can be accessed by the app are strictly limited; you can read App Sandboxing for more information.

Additional Entitlements

Depending on which Electron APIs your app uses, you may need to add additional entitlements to your `parent.plist` file to be able to use these APIs from your app's Mac App Store build.

Network Access

Enable outgoing network connections to allow your app to connect to a server:

```
<key>com.apple.security.network.client</key>
<true/>
```

Enable incoming network connections to allow your app to open a network listening socket:

```
<key>com.apple.security.network.server</key>
<true/>
```

See the Enabling Network Access documentation for more details.

dialog.showOpenDialog

```
<key>com.apple.security.files.user-selected.read-only</key>
<true/>
```

See the Enabling User-Selected File Access documentation for more details.

dialog.showSaveDialog

```
<key>com.apple.security.files.user-selected.read-write</key>
<true/>
```

See the Enabling User-Selected File Access documentation for more details.

Cryptographic Algorithms Used by Electron

Depending on the country and region you are located, Mac App Store may require documenting the cryptographic algorithms used in your app, and even ask you to submit a copy of U.S. Encryption Registration (ERN) approval.

Electron uses following cryptographic algorithms:

- AES - NIST SP 800-38A, NIST SP 800-38D, RFC 3394
- HMAC - FIPS 198-1
- ECDSA - ANS X9.62–2005
- ECDH - ANS X9.63–2001
- HKDF - NIST SP 800-56C
- PBKDF2 - RFC 2898
- RSA - RFC 3447
- SHA - FIPS 180-4
- Blowfish - https://www.schneier.com/cryptography/blowfish/
- CAST - RFC 2144, RFC 2612
- DES - FIPS 46-3
- DH - RFC 2631
- DSA - ANSI X9.30
- EC - SEC 1
- IDEA - "On the Design and Security of Block Ciphers" book by X. Lai
- MD2 - RFC 1319
- MD4 - RFC 6150
- MD5 - RFC 1321
- MDC2 - ISO/IEC 10118-2
- RC2 - RFC 2268
- RC4 - RFC 4345
- RC5 - http://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf
- RIPEMD - ISO/IEC 10118-3

On how to get the ERN approval, you can reference the article: How to legally submit an app to Apple's App Store when it uses encryption (or how to obtain an ERN).

Windows Store Guide

With Windows 8, the good old win32 executable got a new sibling: The Universal Windows Platform. The new `.appx` format does not only enable a number of new powerful APIs like Cortana or Push Notifications, but through the Windows Store, also simplifies installation and updating.

Microsoft developed a tool that compiles Electron apps as `.appx` packages, enabling developers to use some of the goodies found in the new application model. This guide explains how to use it - and what the capabilities and limitations of an Electron AppX package are.

Background and Requirements

Windows 10 "Anniversary Update" is able to run win32 `.exe` binaries by launching them together with a virtualized filesystem and registry. Both are created during compilation by running app and installer inside a Windows Container, allowing Windows to identify exactly which modifications to the operating system are done during installation. Pairing the executable with a virtual filesystem and a virtual registry allows Windows to enable one-click installation and uninstallation.

In addition, the exe is launched inside the appx model - meaning that it can use many of the APIs available to the Universal Windows Platform. To gain even more capabilities, an Electron app can pair up with an invisible UWP background task launched together with the `exe` - sort of launched as a sidekick to run tasks in the background, receive push notifications, or to communicate with other UWP applications.

To compile any existing Electron app, ensure that you have the following requirements:

- Windows 10 with Anniversary Update (released August 2nd, 2016)
- The Windows 10 SDK, downloadable here
- At least Node 4 (to check, run `node -v`)

Then, go and install the `electron-windows-store` CLI:

```
npm install -g electron-windows-store
```

Step 1: Package Your Electron Application

Package the application using electron-packager (or a similar tool). Make sure to remove `node_modules` that you don't need in your final application, since any module you don't actually need will just increase your application's size.

The output should look roughly like this:

```
├── Ghost.exe
├── LICENSE
├── content_resources_200_percent.pak
├── content_shell.pak
├── d3dcompiler_47.dll
├── ffmpeg.dll
├── icudtl.dat
├── libEGL.dll
├── libGLESv2.dll
├── locales
│   ├── am.pak
│   ├── ar.pak
│   ├── [...]
├── natives_blob.bin
├── node.dll
├── resources
│   ├── app
│   └── atom.asar
├── snapshot_blob.bin
├── squirrel.exe
├── ui_resources_200_percent.pak
└── xinput1_3.dll
```

Step 2: Running electron-windows-store

From an elevated PowerShell (run it "as Administrator"), run `electron-windows-store` with the required parameters, passing both the input and output directories, the app's name and version, and confirmation that `node_modules` should be flattened.

```
electron-windows-store `
    --input-directory C:\myelectronapp `
    --output-directory C:\output\myelectronapp `
    --flatten true `
    --package-version 1.0.0.0 `
    --package-name myelectronapp
```

Once executed, the tool goes to work: It accepts your Electron app as an input, flattening the `node_modules`. Then, it archives your application as `app.zip`. Using an installer and a Windows Container, the tool creates an "expanded" AppX package - including the Windows Application Manifest (`AppXManifest.xml`) as well as the virtual file system and the virtual registry inside your output folder.

Once the expanded AppX files are created, the tool uses the Windows App Packager (`MakeAppx.exe`) to create a single-file AppX package from those files on disk. Finally, the tool can be used to create a trusted certificate on your computer to sign the new AppX package. With the signed AppX package, the CLI can also automatically install the package on your machine.

Step 3: Using the AppX Package

In order to run your package, your users will need Windows 10 with the so-called "Anniversary Update" - details on how to update Windows can be found here.

In opposition to traditional UWP apps, packaged apps currently need to undergo a manual verification process, for which you can apply here. In the meantime, all users will be able to just install your package by double-clicking it, so a submission to the store might not be necessary if you're simply looking for an easier installation method. In managed environments (usually enterprises), the `Add-AppxPackage` PowerShell Cmdlet can be used to install it in an automated fashion.

Another important limitation is that the compiled AppX package still contains a win32 executable - and will therefore not run on Xbox, HoloLens, or Phones.

Optional: Add UWP Features using a BackgroundTask

You can pair your Electron app up with an invisible UWP background task that gets to make full use of Windows 10 features - like push notifications, Cortana integration, or live tiles.

To check out how an Electron app that uses a background task to send toast notifications and live tiles, check out the Microsoft-provided sample.

Optional: Convert using Container Virtualization

To generate the AppX package, the `electron-windows-store` CLI uses a template that should work for most Electron apps. However, if you are using a custom installer, or should you experience any trouble with the generated package, you can attempt to create a package using compilation with a Windows Container - in that mode, the CLI will install and run your application in blank Windows Container to determine what modifications your application is exactly doing to the operating system.

Before running the CLI for the, you will have to setup the "Windows Desktop App Converter". This will take a few minutes, but don't worry - you only have to do this once. Download and Desktop App Converter from here. You will receive two files: `DesktopAppConverter.zip` and `BaseImage-14316.wim` .

1. Unzip `DesktopAppConverter.zip` . From an elevated PowerShell (opened with "run as Administrator", ensure that your systems execution policy allows us to run everything we intend to run by calling `Set-ExecutionPolicy bypass` .
2. Then, run the installation of the Desktop App Converter, passing in the location of the Windows base Image (downloaded as `BaseImage-14316.wim`), by calling `.\DesktopAppConverter.ps1 -Setup -BaseImage .\BaseImage-14316.wim` .
3. If running the above command prompts you for a reboot, please restart your machine and run the above command again after a successful restart.

Once installation succeeded, you can move on to compiling your Electron app.

Application Packaging

To mitigate issues around long path names on Windows, slightly speed up `require` and conceal your source code from cursory inspection, you can choose to package your app into an asar archive with little changes to your source code.

Generating `asar` Archive

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.

Steps to package your app into an `asar` archive:

1. Install the asar Utility

```
$ npm install -g asar
```

2. Package with `asar pack`

```
$ asar pack your-app app.asar
```

Using `asar` Archives

In Electron there are two sets of APIs: Node APIs provided by Node.js and Web APIs provided by Chromium. Both APIs support reading files from `asar` archives.

Node API

With special patches in Electron, Node APIs like `fs.readFile` and `require` treat `asar` archives as virtual directories, and the files in it as normal files in the filesystem.

For example, suppose we have an `example.asar` archive under `/path/to` :

```
$ asar list /path/to/example.asar
/app.js
/file.txt
/dir/module.js
/static/index.html
/static/main.css
/static/jquery.min.js
```

Read a file in the `asar` archive:

```
const fs = require('fs')
fs.readFileSync('/path/to/example.asar/file.txt')
```

List all files under the root of the archive:

```
const fs = require('fs')
fs.readdirSync('/path/to/example.asar')
```

Use a module from the archive:

```
require('/path/to/example.asar/dir/module.js')
```

You can also display a web page in an `asar` archive with `BrowserWindow`:

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow({width: 800, height: 600})
win.loadURL('file:///path/to/example.asar/static/index.html')
```

Web API

In a web page, files in an archive can be requested with the `file:` protocol. Like the Node API, `asar` archives are treated as directories.

For example, to get a file with `$.get`:

```
<script>
let $ = require('./jquery.min.js')
$.get('file:///path/to/example.asar/file.txt', (data) => {
  console.log(data)
})
</script>
```

Treating an `asar` Archive as a Normal File

For some cases like verifying the `asar` archive's checksum, we need to read the content of an `asar` archive as a file. For this purpose you can use the built-in `original-fs` module which provides original `fs` APIs without `asar` support:

```
const originalFs = require('original-fs')
originalFs.readFileSync('/path/to/example.asar')
```

You can also set `process.noAsar` to `true` to disable the support for `asar` in the `fs` module:

```
const fs = require('fs')
process.noAsar = true
fs.readFileSync('/path/to/example.asar')
```

Limitations of the Node API

Even though we tried hard to make `asar` archives in the Node API work like directories as much as possible, there are still limitations due to the low-level nature of the Node API.

Archives Are Read-only

The archives can not be modified so all Node APIs that can modify files will not work with `asar` archives.

Working Directory Can Not Be Set to Directories in Archive

Though `asar` archives are treated as directories, there are no actual directories in the filesystem, so you can never set the working directory to directories in `asar` archives. Passing them as the `cwd` option of some APIs will also cause errors.

Extra Unpacking on Some APIs

Most `fs` APIs can read a file or get a file's information from `asar` archives without unpacking, but for some APIs that rely on passing the real file path to underlying system calls, Electron will extract the needed file into a temporary file and pass the path of the temporary file to the APIs to make them work. This adds a little overhead for those APIs.

APIs that requires extra unpacking are:

- `child_process.execFile`
- `child_process.execFileSync`
- `fs.open`
- `fs.openSync`
- `process.dlopen` - Used by `require` on native modules

Fake Stat Information of `fs.stat`

The `Stats` object returned by `fs.stat` and its friends on files in `asar` archives is generated by guessing, because those files do not exist on the filesystem. So you should not trust the `Stats` object except for getting file size and checking file type.

Executing Binaries Inside `asar` Archive

There are Node APIs that can execute binaries like `child_process.exec` , `child_process.spawn` and `child_process.execFile` , but only `execFile` is supported to execute binaries inside `asar` archive.

This is because `exec` and `spawn` accept `command` instead of `file` as input, and `command`s are executed under shell. There is no reliable way to determine whether a command uses a file in asar archive, and even if we do, we can not be sure whether we can replace the path in command without side effects.

Adding Unpacked Files in `asar` Archive

As stated above, some Node APIs will unpack the file to filesystem when calling, apart from the performance issues, it could also lead to false alerts of virus scanners.

To work around this, you can unpack some files creating archives by using the `--unpack` option, an example of excluding shared libraries of native modules is:

```
$ asar pack app app.asar --unpack *.node
```

After running the command, apart from the `app.asar` , there is also an `app.asar.unpacked` folder generated which contains the unpacked files, you should copy it together with `app.asar` when shipping it to users.

Using Native Node Modules

The 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.

How to install native modules

Three ways to install native modules:

Using `npm`

By setting a few environment variables, you can use `npm` to install modules directly.

An example of installing all dependencies for Electron:

```
# Electron's version.
export npm_config_target=1.2.3
# The architecture of Electron, can be ia32 or x64.
export npm_config_arch=x64
export npm_config_target_arch=x64
# Download headers for Electron.
export npm_config_disturl=https://atom.io/download/electron
# Tell node-pre-gyp that we are building for Electron.
export npm_config_runtime=electron
# Tell node-pre-gyp to build module from source code.
export npm_config_build_from_source=true
# Install all dependencies, and store cache to ~/.electron-gyp.
HOME=~/.electron-gyp npm install
```

Installing modules and rebuilding for Electron

You can also choose to install modules like other Node projects, and then rebuild the modules for Electron with the `electron-rebuild` package. This module can get the version of Electron and handle the manual steps of downloading headers and building native modules for your app.

An example of installing `electron-rebuild` and then rebuild modules with it:

```
npm install --save-dev electron-rebuild

# Every time you run "npm install", run this:
./node_modules/.bin/electron-rebuild

# On Windows if you have trouble, try:
.\node_modules\.bin\electron-rebuild.cmd
```

Manually building for Electron

If you are a developer developing a native module and want to test it against Electron, you might want to rebuild the module for Electron manually. You can use `node-gyp` directly to build for Electron:

```
cd /path-to-module/
HOME=~/.electron-gyp node-gyp rebuild --target=1.2.3 --arch=x64 --dist-url=https://atom.io/download/electron
```

The `HOME=~/.electron-gyp` changes where to find development headers. The `--target=1.2.3` is version of Electron. The `--dist-url=...` specifies where to download the headers. The `--arch=x64` says the module is built for 64bit system.

Troubleshooting

If you installed a native module and found it was not working, you need to check following things:

- The architecture of module has to match Electron's architecture (ia32 or x64).
- After you upgraded Electron, you usually need to rebuild the modules.
- When in doubt, run `electron-rebuild` first.

Modules that rely on `prebuild`

`prebuild` provides a way to easily publish native Node modules with prebuilt binaries for multiple versions of Node and Electron.

If modules provide binaries for the usage in Electron, make sure to omit `--build-from-source` and the `npm_config_build_from_source` environment variable in order to take full advantage of the prebuilt binaries.

Modules that rely on `node-pre-gyp`

The `node-pre-gyp` tool provides a way to deploy native Node modules with prebuilt binaries, and many popular modules are using it.

Usually those modules work fine under Electron, but sometimes when Electron uses a newer version of V8 than Node, and there are ABI changes, bad things may happen. So in general it is recommended to always build native modules from source code.

If you are following the `npm` way of installing modules, then this is done by default, if not, you have to pass `--build-from-source` to `npm`, or set the `npm_config_build_from_source` environment variable.

Debugging the Main Process

The DevTools in an Electron browser window can only debug JavaScript that's executed in that window (i.e. the web pages). To debug JavaScript that's executed in the main process you will need to use an external debugger and launch Electron with the `--debug` or `--debug-brk` switch.

Command Line Switches

Use one of the following command line switches to enable debugging of the main process:

`--debug=[port]`

Electron will listen for V8 debugger protocol messages on the specified `port`, an external debugger will need to connect on this port. The default `port` is `5858`.

```
electron --debug=5858 your/app
```

`--debug-brk=[port]`

Like `--debug` but pauses execution on the first line of JavaScript.

External Debuggers

You will need to use a debugger that supports the V8 debugger protocol, the following guides should help you to get started:

- Debugging the Main Process in VSCode
- Debugging the Main Process in node-inspector

Using Selenium and WebDriver

From ChromeDriver - WebDriver for Chrome:

> WebDriver is an open source tool for automated testing of web apps across many browsers. It provides capabilities for navigating to web pages, user input, JavaScript execution, and more. ChromeDriver is a standalone server which implements WebDriver's wire protocol for Chromium. It is being developed by members of the Chromium and WebDriver teams.

Setting up Spectron

Spectron is the officially supported ChromeDriver testing framework for Electron. It is built on top of WebdriverIO and has helpers to access Electron APIs in your tests and bundles ChromeDriver.

```
$ npm install --save-dev spectron
```

```
// A simple test to verify a visible window is opened with a title
var Application = require('spectron').Application
var assert = require('assert')

var app = new Application({
  path: '/Applications/MyApp.app/Contents/MacOS/MyApp'
})

app.start().then(function () {
  // Check if the window is visible
  return app.browserWindow.isVisible()
}).then(function (isVisible) {
  // Verify the window is visible
  assert.equal(isVisible, true)
}).then(function () {
  // Get the window's title
  return app.client.getTitle()
}).then(function (title) {
  // Verify the window's title
  assert.equal(title, 'My App')
}).catch(function (error) {
  // Log any failures
  console.error('Test failed', error.message)
}).then(function () {
  // Stop the application
  return app.stop()
})
```

Setting up with WebDriverJs

WebDriverJs provides a Node package for testing with web driver, we will use it as an example.

1. Start ChromeDriver

First you need to download the `chromedriver` binary, and run it:

```
$ npm install electron-chromedriver
$ ./node_modules/.bin/chromedriver
Starting ChromeDriver (v2.10.291558) on port 9515
Only local connections are allowed.
```

Remember the port number `9515` , which will be used later

2. Install WebDriverJS

```
$ npm install selenium-webdriver
```

3. Connect to ChromeDriver

The usage of `selenium-webdriver` with Electron is basically the same with upstream, except that you have to manually specify how to connect chrome driver and where to find Electron's binary:

```
const webdriver = require('selenium-webdriver')

const driver = new webdriver.Builder()
  // The "9515" is the port opened by chrome driver.
  .usingServer('http://localhost:9515')
  .withCapabilities({
    chromeOptions: {
      // Here is the path to your Electron binary.
      binary: '/Path-to-Your-App.app/Contents/MacOS/Electron'
    }
  })
  .forBrowser('electron')
  .build()

driver.get('http://www.google.com')
driver.findElement(webdriver.By.name('q')).sendKeys('webdriver')
driver.findElement(webdriver.By.name('btnG')).click()
driver.wait(() => {
  return driver.getTitle().then((title) => {
    return title === 'webdriver - Google Search'
  })
}, 1000)

driver.quit()
```

Setting up with WebdriverIO

WebdriverIO provides a Node package for testing with web driver.

1. Start ChromeDriver

First you need to download the `chromedriver` binary, and run it:

```
$ npm install electron-chromedriver
$ ./node_modules/.bin/chromedriver --url-base=wd/hub --port=9515
Starting ChromeDriver (v2.10.291558) on port 9515
Only local connections are allowed.
```

Remember the port number `9515` , which will be used later

2. Install WebdriverIO

```
$ npm install webdriverio
```

3. Connect to chrome driver

```
const webdriverio = require('webdriverio')
const options = {
  host: 'localhost', // Use localhost as chrome driver server
  port: 9515,        // "9515" is the port opened by chrome driver.
  desiredCapabilities: {
    browserName: 'chrome',
    chromeOptions: {
      binary: '/Path-to-Your-App/electron', // Path to your Electron binary.
      args: [/* cli arguments */]           // Optional, perhaps 'app=' + /path/to/your/app/
    }
  }
}

let client = webdriverio.remote(options)

client
  .init()
  .url('http://google.com')
  .setValue('#q', 'webdriverio')
  .click('#btnG')
  .getTitle().then((title) => {
    console.log('Title was: ' + title)
  })
  .end()
```

Workflow

To test your application without rebuilding Electron, simply place your app source into Electron's resource directory.

Alternatively, pass an argument to run with your electron binary that points to your app's folder. This eliminates the need to copy-paste your app into Electron's resource directory.

DevTools Extension

Electron supports the Chrome DevTools Extension, which can be used to extend the ability of devtools for debugging popular web frameworks.

How to load a DevTools Extension

This document outlines the process for manually loading an extension. You may also try electron-devtools-installer, a third-party tool that downloads extensions directly from the Chrome WebStore.

To load an extension in Electron, you need to download it in Chrome browser, locate its filesystem path, and then load it by calling the `BrowserWindow.addDevToolsExtension(extension)` API.

Using the React Developer Tools as example:

1. Install it in Chrome browser.
2. Navigate to `chrome://extensions`, and find its extension ID, which is a hash string like `fmkadmapgofadopljbjfkapdkoienihi`.
3. Find out filesystem location used by Chrome for storing extensions:
   - on Windows it is `%LOCALAPPDATA%\Google\Chrome\User Data\Default\Extensions`;
   - on Linux it could be:
     - `~/.config/google-chrome/Default/Extensions/`
     - `~/.config/google-chrome-beta/Default/Extensions/`
     - `~/.config/google-chrome-canary/Default/Extensions/`
     - `~/.config/chromium/Default/Extensions/`
   - on macOS it is `~/Library/Application Support/Google/Chrome/Default/Extensions`.
4. Pass the location of the extension to `BrowserWindow.addDevToolsExtension` API, for the React Developer Tools, it is something like: `~/Library/Application Support/Google/Chrome/Default/Extensions/fmkadmapgofadopljbjfkapdkoienihi/0.15.0_0`

Note: The `BrowserWindow.addDevToolsExtension` API cannot be called before the ready event of the app module is emitted.

The name of the extension is returned by `BrowserWindow.addDevToolsExtension`, and you can pass the name of the extension to the `BrowserWindow.removeDevToolsExtension` API to unload it.

Supported DevTools Extensions

Electron only supports a limited set of `chrome.*` APIs, so some extensions using unsupported `chrome.*` APIs for chrome extension features may not work. Following Devtools Extensions are tested and guaranteed to work in Electron:

- Ember Inspector
- React Developer Tools
- Backbone Debugger
- jQuery Debugger
- AngularJS Batarang
- Vue.js devtools
- Cerebral Debugger
- Redux DevTools Extension

What should I do if a DevTools Extension is not working?

First please make sure the extension is still being maintained, some extensions can not even work for recent versions of Chrome browser, and we are not able to do anything for them.

Then file a bug at Electron's issues list, and describe which part of the extension is not working as expected.

Using Pepper Flash Plugin

Electron supports the Pepper Flash plugin. To use the Pepper Flash plugin in Electron, you should manually specify the location of the Pepper Flash plugin and then enable it in your application.

Prepare a Copy of Flash Plugin

On macOS and Linux, the details of the Pepper Flash plugin can be found by navigating to `chrome://plugins` in the Chrome browser. Its location and version are useful for Electron's Pepper Flash support. You can also copy it to another location.

Add Electron Switch

You can directly add `--ppapi-flash-path` and `--ppapi-flash-version` to the Electron command line or by using the `app.commandLine.appendSwitch` method before the app ready event. Also, turn on `plugins` option of `BrowserWindow`.

For example:

```
const {app, BrowserWindow} = require('electron')
const path = require('path')

// Specify flash path, supposing it is placed in the same directory with main.js.
let pluginName
switch (process.platform) {
  case 'win32':
    pluginName = 'pepflashplayer.dll'
    break
  case 'darwin':
    pluginName = 'PepperFlashPlayer.plugin'
    break
  case 'linux':
    pluginName = 'libpepflashplayer.so'
    break
}
app.commandLine.appendSwitch('ppapi-flash-path', path.join(__dirname, pluginName))

// Optional: Specify flash version, for example, v17.0.0.169
app.commandLine.appendSwitch('ppapi-flash-version', '17.0.0.169')

app.on('ready', () => {
  let win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      plugins: true
    }
  })
  win.loadURL(`file://${__dirname}/index.html`)
  // Something else
})
```

You can also try loading the system wide Pepper Flash plugin instead of shipping the plugins yourself, its path can be received by calling `app.getPath('pepperFlashSystemPlugin')`.

Enable Flash Plugin in a `<webview>` Tag

Add `plugins` attribute to `<webview>` tag.

```
<webview src="http://www.adobe.com/software/flash/about/" plugins></webview>
```

Troubleshooting

You can check if Pepper Flash plugin was loaded by inspecting `navigator.plugins` in the console of devtools (although you can't know if the plugin's path is correct).

The architecture of Pepper Flash plugin has to match Electron's one. On Windows, a common error is to use 32bit version of Flash plugin against 64bit version of Electron.

On Windows the path passed to `--ppapi-flash-path` has to use `\` as path delimiter, using POSIX-style paths will not work.

For some operations, such as streaming media using RTMP, it is necessary to grant wider permissions to players' `.swf` files. One way of accomplishing this, is to use nw-flash-trust.

Using Widevine CDM Plugin

In Electron you can use the Widevine CDM plugin shipped with Chrome browser.

Getting the plugin

Electron doesn't ship with the Widevine CDM plugin for license reasons, to get it, you need to install the official Chrome browser first, which should match the architecture and Chrome version of the Electron build you use.

Note: The major version of Chrome browser has to be the same with the Chrome version used by Electron, otherwise the plugin will not work even though `navigator.plugins` would show it has been loaded.

Windows & macOS

Open `chrome://components/` in Chrome browser, find `WidevineCdm` and make sure it is up to date, then you can find all the plugin binaries from the `APP_DATA/Google/Chrome/WidevineCDM/VERSION/_platform_specific/PLATFORM_ARCH/` directory.

`APP_DATA` is system's location for storing app data, on Windows it is `%LOCALAPPDATA%` , on macOS it is `~/Library/Application Support` . `VERSION` is Widevine CDM plugin's version string, like `1.4.8.866` . `PLATFORM` is `mac` or `win` . `ARCH` is `x86` or `x64` .

On Windows the required binaries are `widevinecdm.dll` and `widevinecdmadapter.dll` , on macOS they are `libwidevinecdm.dylib` and `widevinecdmadapter.plugin` . You can copy them to anywhere you like, but they have to be put together.

Linux

On Linux the plugin binaries are shipped together with Chrome browser, you can find them under `/opt/google/chrome` , the filenames are `libwidevinecdm.so` and `libwidevinecdmadapter.so` .

Using the plugin

After getting the plugin files, you should pass the `widevinecdmadapter` 's path to Electron with `--widevine-cdm-path` command line switch, and the plugin's version with `--widevine-cdm-version` switch.

Note: Though only the `widevinecdmadapter` binary is passed to Electron, the `widevinecdm` binary has to be put aside it.

The command line switches have to be passed before the `ready` event of `app` module gets emitted, and the page that uses this plugin must have plugin enabled.

Example code:

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

// You have to pass the filename of `widevinecdmadapter` here, it is
// * `widevinecdmadapter.plugin` on macOS,
// * `libwidevinecdmadapter.so` on Linux,
// * `widevinecdmadapter.dll` on Windows.
app.commandLine.appendSwitch('widevine-cdm-path', '/path/to/widevinecdmadapter.plugin')
// The version of plugin can be got from `chrome://plugins` page in Chrome.
app.commandLine.appendSwitch('widevine-cdm-version', '1.4.8.866')

let win = null
app.on('ready', () => {
  win = new BrowserWindow({
    webPreferences: {
      // The `plugins` have to be enabled.
      plugins: true
    }
  })
  win.show()
})
```

Verifying the plugin

To verify whether the plugin works, you can use following ways:

- Open devtools and check whether `navigator.plugins` includes the Widevine CDM plugin.
- Open https://shaka-player-demo.appspot.com/ and load a manifest that uses `Widevine`.
- Open http://www.dash-player.com/demo/drm-test-area/, check whether the page says `bitdash uses Widevine in your browser`, then play the video.

Testing on Headless CI Systems (Travis CI, Jenkins)

Being based on Chromium, Electron requires a display driver to function. If Chromium can't find a display driver, Electron will simply fail to launch - and therefore not executing any of your tests, regardless of how you are running them. Testing Electron-based apps on Travis, Circle, Jenkins or similar Systems requires therefore a little bit of configuration. In essence, we need to use a virtual display driver.

Configuring the Virtual Display Server

First, install Xvfb. It's a virtual framebuffer, implementing the X11 display server protocol - it performs all graphical operations in memory without showing any screen output, which is exactly what we need.

Then, create a virtual xvfb screen and export an environment variable called DISPLAY that points to it. Chromium in Electron will automatically look for `$DISPLAY`, so no further configuration of your app is required. This step can be automated with Paul Betts's xvfb-maybe: Prepend your test commands with `xvfb-maybe` and the little tool will automatically configure xvfb, if required by the current system. On Windows or macOS, it will simply do nothing.

```
## On Windows or macOS, this just invokes electron-mocha
## On Linux, if we are in a headless environment, this will be equivalent
## to xvfb-run electron-mocha ./test/*.js
xvfb-maybe electron-mocha ./test/*.js
```

Travis CI

On Travis, your `.travis.yml` should look roughly like this:

```
addons:
  apt:
    packages:
      - xvfb

install:
  - export DISPLAY=':99.0'
  - Xvfb :99 -screen 0 1024x768x24 > /dev/null 2>&1 &
```

Jenkins

For Jenkins, a Xvfb plugin is available.

Circle CI

Circle CI is awesome and has xvfb and `$DISPLAY` already setup, so no further configuration is required.

AppVeyor

AppVeyor runs on Windows, supporting Selenium, Chromium, Electron and similar tools out of the box - no configuration is required.

Offscreen Rendering

Offscreen rendering lets you obtain the content of a browser window in a bitmap, so it can be rendered anywhere, for example on a texture in a 3D scene. The offscreen rendering in Electron uses a similar approach than the Chromium Embedded Framework project.

Two modes of rendering can be used and only the dirty area is passed in the `'paint'` event to be more efficient. The rendering can be stopped, continued and the frame rate can be set. The specified frame rate is a top limit value, when there is nothing happening on a webpage, no frames are generated. The maximum frame rate is 60, because above that there is no benefit, just performance loss.

Note: An offscreen window is always created as a Frameless Window.

Two modes of rendering

GPU accelerated

GPU accelerated rendering means that the GPU is used for composition. Because of that the frame has to be copied from the GPU which requires more performance, thus this mode is quite a bit slower than the other one. The benefit of this mode that WebGL and 3D CSS animations are supported.

Software output device

This mode uses a software output device for rendering in the CPU, so the frame generation is much faster, thus this mode is preferred over the GPU accelerated one.

To enable this mode GPU acceleration has to be disabled by calling the `app.disableHardwareAcceleration()` API.

Usage

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

app.disableHardwareAcceleration()

let win
app.once('ready', () => {
  win = new BrowserWindow({
    webPreferences: {
      offscreen: true
    }
  })
  win.loadURL('http://github.com')
  win.webContents.on('paint', (event, dirty, image) => {
    // updateBitmap(dirty, image.getBitmap())
  })
  win.webContents.setFrameRate(30)
})
```

Quick Start

Electron enables you to create desktop applications with pure JavaScript by providing a runtime with rich native (operating system) APIs. You could see it as a variant of the Node.js runtime that is focused on desktop applications instead of web servers.

This doesn't mean Electron is a JavaScript binding to graphical user interface (GUI) libraries. Instead, Electron uses web pages as its GUI, so you could also see it as a minimal Chromium browser, controlled by JavaScript.

Main Process

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.

Renderer Process

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.

In Electron, we have several ways to communicate between the main process and renderer processes. Like `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.

Write your First Electron App

Generally, an Electron app is structured like this:

```
your-app/
├── package.json
├── main.js
└── index.html
```

The format of `package.json` is exactly the same as that of Node's modules, and the script specified by the `main` field is the startup script of your app, which will run the main process. An example of your `package.json` might look like this:

```
{
  "name"    : "your-app",
  "version" : "0.1.0",
  "main"    : "main.js"
}
```

Note: If the `main` field is not present in `package.json`, Electron will attempt to load an `index.js`.

The `main.js` should create windows and handle system events, a typical example being:

```
const {app, BrowserWindow} = require('electron')
const path = require('path')
const url = require('url')

// Keep a global reference of the window object, if you don't, the window will
// be closed automatically when the JavaScript object is garbage collected.
let win

function createWindow () {
  // Create the browser window.
  win = new BrowserWindow({width: 800, height: 600})

  // and load the index.html of the app.
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))

  // Open the DevTools.
  win.webContents.openDevTools()

  // Emitted when the window is closed.
  win.on('closed', () => {
    // Dereference the window object, usually you would store windows
    // in an array if your app supports multi windows, this is the time
    // when you should delete the corresponding element.
    win = null
  })
}

// This method will be called when Electron has finished
// initialization and is ready to create browser windows.
// Some APIs can only be used after this event occurs.
app.on('ready', createWindow)

// Quit when all windows are closed.
app.on('window-all-closed', () => {
  // On macOS it is common for applications and their menu bar
  // to stay active until the user quits explicitly with Cmd + Q
  if (process.platform !== 'darwin') {
    app.quit()
  }
})

app.on('activate', () => {
  // On macOS it's common to re-create a window in the app when the
  // dock icon is clicked and there are no other windows open.
  if (win === null) {
    createWindow()
  }
})

// In this file you can include the rest of your app's specific main process
// code. You can also put them in separate files and require them here.
```

Finally the `index.html` is the web page you want to show:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    We are using node <script>document.write(process.versions.node)</script>,
    Chrome <script>document.write(process.versions.chrome)</script>,
    and Electron <script>document.write(process.versions.electron)</script>.
  </body>
</html>
```

Run your app

Once you've created your initial `main.js`, `index.html`, and `package.json` files, you'll probably want to try running your app locally to test it and make sure it's working as expected.

`electron`

`electron` is an `npm` module that contains pre-compiled versions of Electron.

If you've installed it globally with `npm`, then you will only need to run the following in your app's source directory:

```
electron .
```

If you've installed it locally, then run:

macOS / Linux

```
$ ./node_modules/.bin/electron .
```

Windows

```
$ .\node_modules\.bin\electron .
```

Manually Downloaded Electron Binary

If you downloaded Electron manually, you can also use the included binary to execute your app directly.

Windows

```
$ .\electron\electron.exe your-app\
```

Linux

```
$ ./electron/electron your-app/
```

macOS

```
$ ./Electron.app/Contents/MacOS/Electron your-app/
```

`Electron.app` here is part of the Electron's release package, you can download it from here.

Run as a distribution

After you're done writing your app, you can create a distribution by following the Application Distribution guide and then executing the packaged app.

Try this Example

Clone and run the code in this tutorial by using the `electron/electron-quick-start` repository.

Note: Running this requires Git and Node.js (which includes npm) on your system.

```
# Clone the repository
$ git clone https://github.com/electron/electron-quick-start
# Go into the repository
$ cd electron-quick-start
# Install dependencies
$ npm install
# Run the app
$ npm start
```

For more example apps, see the list of boilerplates created by the awesome electron community.

Desktop Environment Integration

Different operating systems provide different features for integrating desktop applications into their desktop environments. For example, on Windows, applications can put shortcuts in the JumpList of task bar, and on Mac, applications can put a custom menu in the dock menu.

This guide explains how to integrate your application into those desktop environments with Electron APIs.

Notifications (Windows, Linux, macOS)

All three operating systems provide means for applications to send notifications to the user. Electron conveniently allows developers to send notifications with the HTML5 Notification API, using the currently running operating system's native notification APIs to display it.

Note: Since this is an HTML5 API it is only available in the renderer process.

```
let myNotification = new Notification('Title', {
  body: 'Lorem Ipsum Dolor Sit Amet'
})

myNotification.onclick = () => {
  console.log('Notification clicked')
}
```

While code and user experience across operating systems are similar, there are fine differences.

Windows

- On Windows 10, notifications "just work".
- On Windows 8.1 and Windows 8, a shortcut to your app, with a Application User Model ID, must be installed to the Start screen. Note, however, that it does not need to be pinned to the Start screen.
- On Windows 7, notifications are not supported. You can however send "balloon notifications" using the Tray API.

Furthermore, the maximum length for the notification body is 250 characters, with the Windows team recommending that notifications should be kept to 200 characters.

Linux

Notifications are sent using `libnotify`, it can show notifications on any desktop environment that follows Desktop Notifications Specification, including Cinnamon, Enlightenment, Unity, GNOME, KDE.

macOS

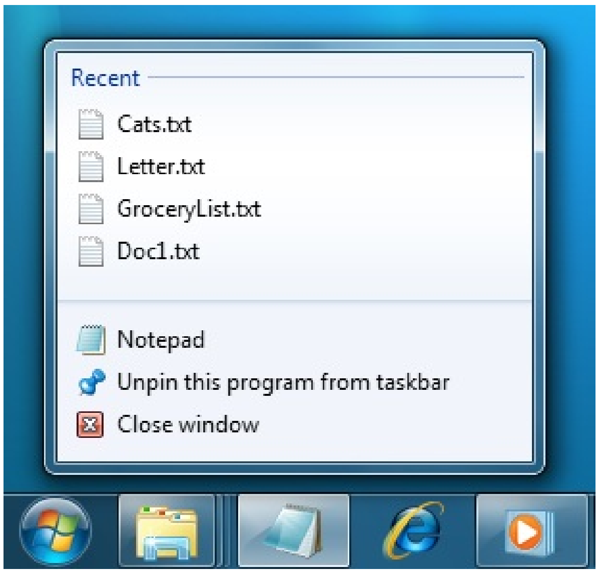
Notifications are straight-forward on macOS, you should however be aware of Apple's Human Interface guidelines regarding notifications.

Note that notifications are limited to 256 bytes in size - and will be truncated if you exceed that limit.

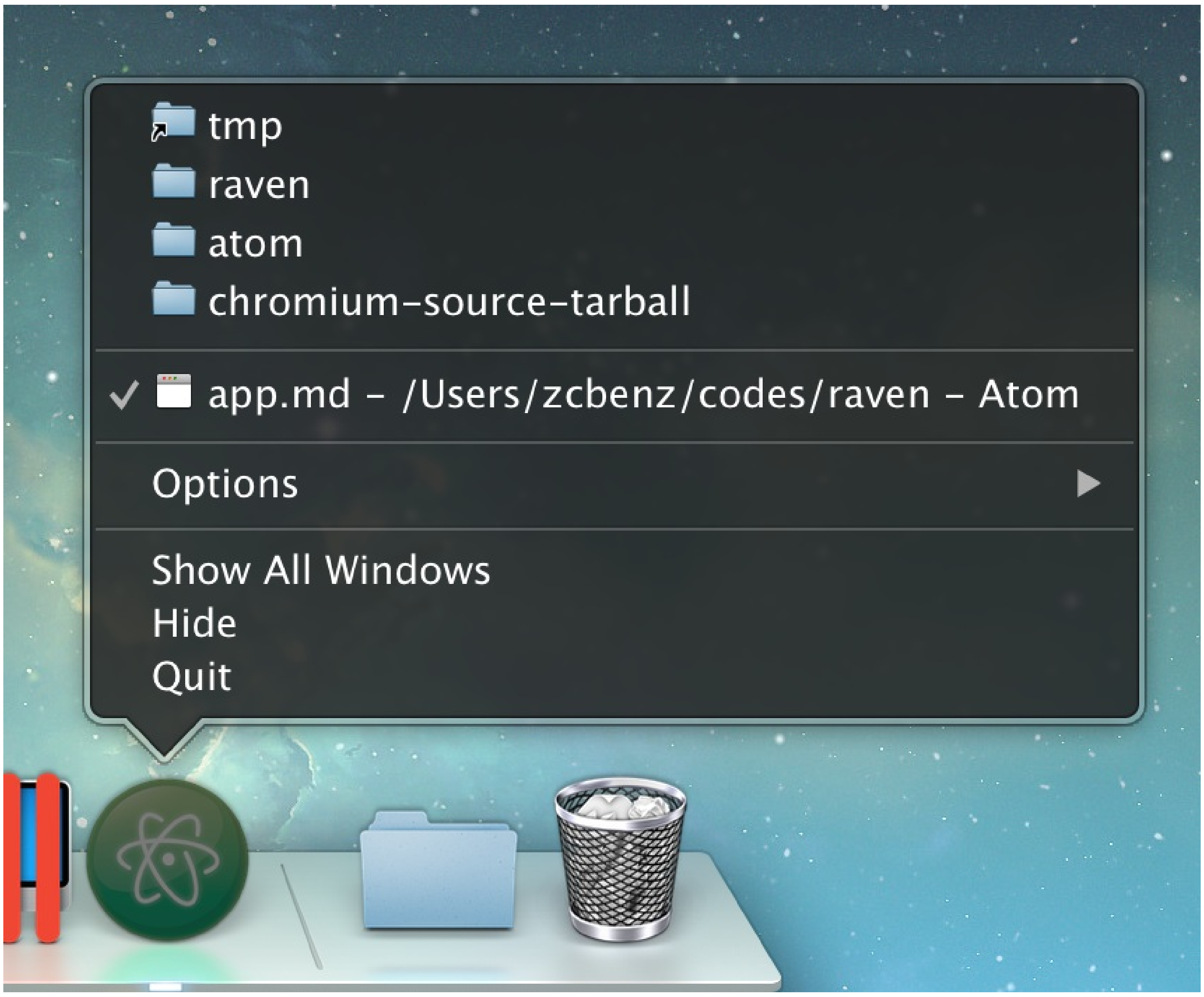
Recent documents (Windows & macOS)

Windows and macOS provide easy access to a list of recent documents opened by the application via JumpList or dock menu, respectively.

JumpList:



Application dock menu:



To add a file to recent documents, you can use the app.addRecentDocument API:

```
const {app} = require('electron')
app.addRecentDocument('/Users/USERNAME/Desktop/work.type')
```

And you can use app.clearRecentDocuments API to empty the recent documents list:

```
const {app} = require('electron')
app.clearRecentDocuments()
```

Windows Notes

In order to be able to use this feature on Windows, your application has to be registered as a handler of the file type of the document, otherwise the file won't appear in JumpList even after you have added it. You can find everything on registering your application in Application Registration.

When a user clicks a file from the JumpList, a new instance of your application will be started with the path of the file added as a command line argument.

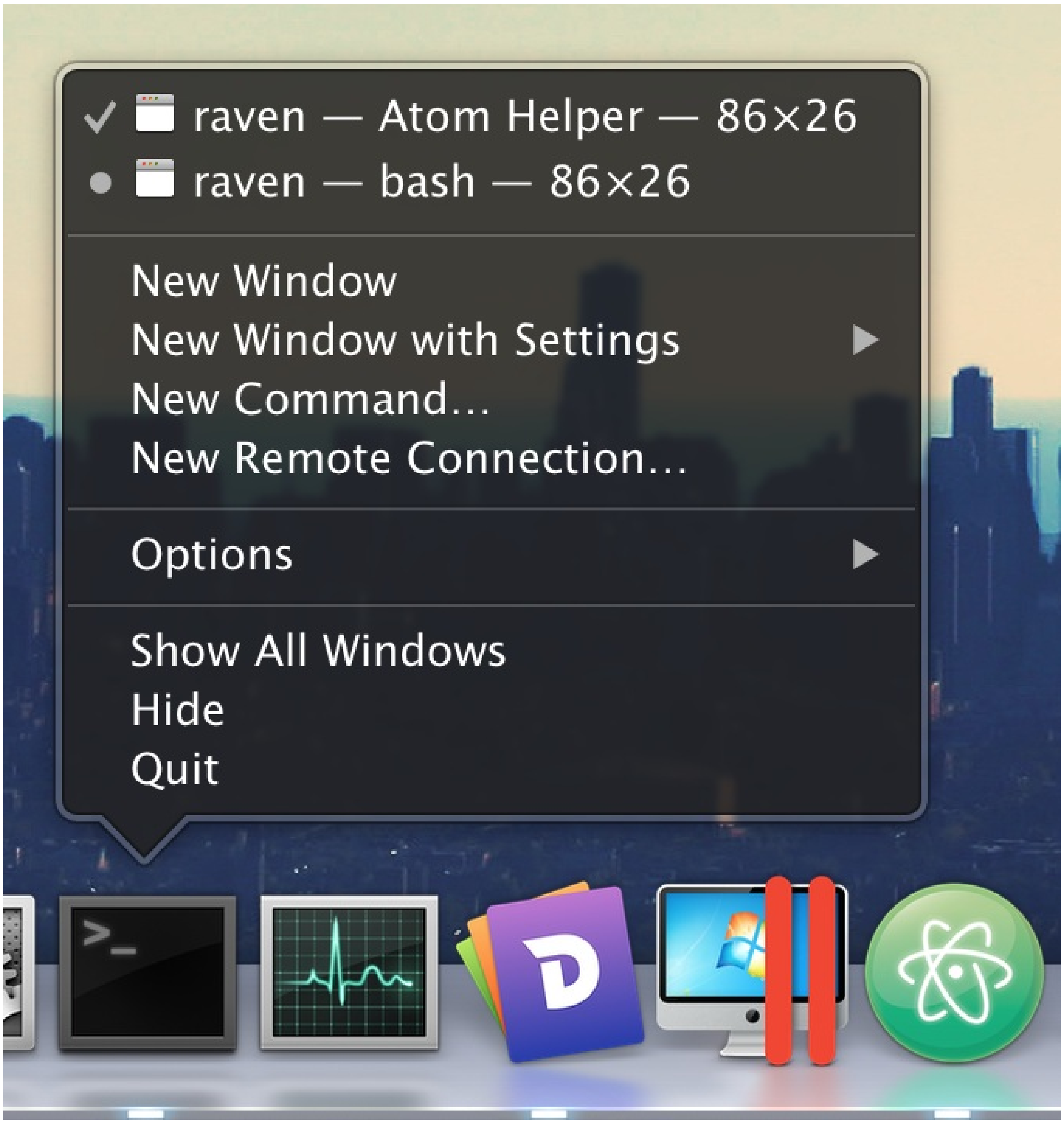
macOS Notes

When a file is requested from the recent documents menu, the `open-file` event of `app` module will be emitted for it.

Custom Dock Menu (macOS)

macOS enables developers to specify a custom menu for the dock, which usually contains some shortcuts for commonly used features of your application:

Dock menu of Terminal.app:



To set your custom dock menu, you can use the `app.dock.setMenu` API, which is only available on macOS:

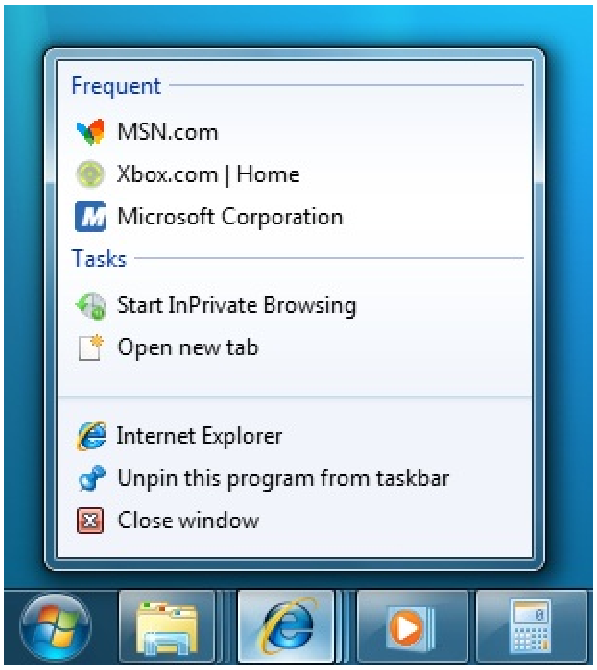
```
const {app, Menu} = require('electron')

const dockMenu = Menu.buildFromTemplate([
  {label: 'New Window', click () { console.log('New Window') }},
  {label: 'New Window with Settings',
    submenu: [
      {label: 'Basic'},
      {label: 'Pro'}
    ]
  },
  {label: 'New Command...'}
])
app.dock.setMenu(dockMenu)
```

User Tasks (Windows)

On Windows you can specify custom actions in the `Tasks` category of JumpList, as quoted from MSDN:

> Applications define tasks based on both the program's features and the key things a user is expected to do with them. Tasks should be context-free, in that the application does not need to be running for them to work. They should also be the statistically most common actions that a normal user would perform in an application, such as compose an email message or open the calendar in a mail program, create a new document in a word processor, launch an application in a certain mode, or launch one of its subcommands. An application should not clutter the menu with advanced features that standard users won't need or one-time actions such as registration. Do not use tasks for promotional items such as upgrades or special offers.
>
> It is strongly recommended that the task list be static. It should remain the same regardless of the state or status of the application. While it is possible to vary the list dynamically, you should consider that this could confuse the user who does not expect that portion of the destination list to change.

Tasks of Internet Explorer:



Unlike the dock menu in macOS which is a real menu, user tasks in Windows work like application shortcuts such that when user clicks a task, a program will be executed with specified arguments.

To set user tasks for your application, you can use app.setUserTasks API:

```
const {app} = require('electron')
app.setUserTasks([
  {
    program: process.execPath,
    arguments: '--new-window',
    iconPath: process.execPath,
    iconIndex: 0,
    title: 'New Window',
    description: 'Create a new window'
  }
])
```

To clean your tasks list, just call `app.setUserTasks` with an empty array:

```
const {app} = require('electron')
app.setUserTasks([])
```

The user tasks will still show even after your application closes, so the icon and program path specified for a task should exist until your application is uninstalled.

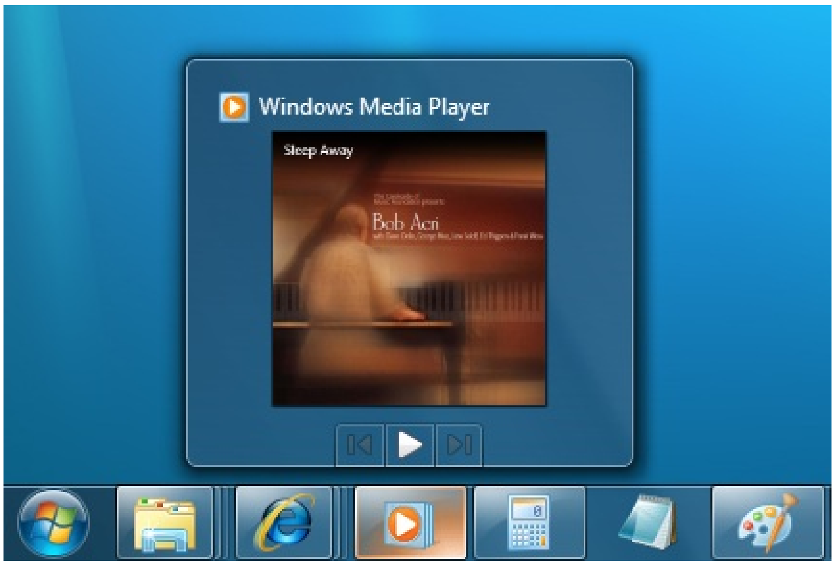
Thumbnail Toolbars

On Windows you can add a thumbnail toolbar with specified buttons in a taskbar layout of an application window. It provides users a way to access to a particular window's command without restoring or activating the window.

From MSDN, it's illustrated:

> This toolbar is simply the familiar standard toolbar common control. It has a maximum of seven buttons. Each button's ID, image, tooltip, and state are defined in a structure, which is then passed to the taskbar. The application can show, enable, disable, or hide buttons from the thumbnail toolbar as required by its current state.
>
> For example, Windows Media Player might offer standard media transport controls such as play, pause, mute, and stop.

Thumbnail toolbar of Windows Media Player:



You can use BrowserWindow.setThumbarButtons to set thumbnail toolbar in your application:

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

let win = new BrowserWindow({
  width: 800,
  height: 600
})

win.setThumbarButtons([
  {
    tooltip: 'button1',
    icon: path.join(__dirname, 'button1.png'),
    click () { console.log('button1 clicked') }
  },
  {
    tooltip: 'button2',
    icon: path.join(__dirname, 'button2.png'),
    flags: ['enabled', 'dismissonclick'],
    click () { console.log('button2 clicked.') }
  }
])
```

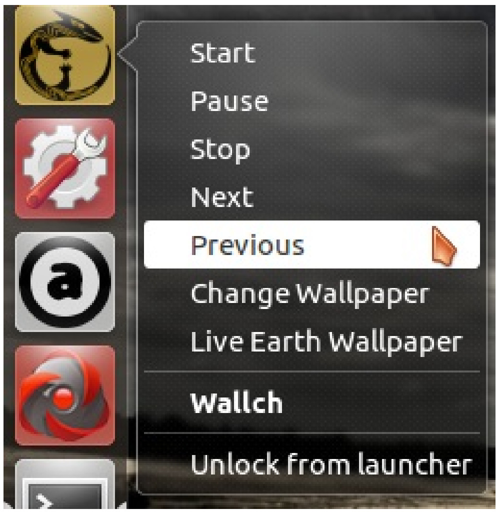
To clean thumbnail toolbar buttons, just call `BrowserWindow.setThumbarButtons` with an empty array:

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()
win.setThumbarButtons([])
```

Unity Launcher Shortcuts (Linux)

In Unity, you can add custom entries to its launcher via modifying the `.desktop` file, see Adding Shortcuts to a Launcher.

Launcher shortcuts of Audacious:



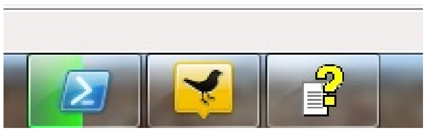
Progress Bar in Taskbar (Windows, macOS, Unity)

On Windows a taskbar button can be used to display a progress bar. This enables a window to provide progress information to the user without the user having to switch to the window itself.

On macOS the progress bar will be displayed as a part of the dock icon.

The Unity DE also has a similar feature that allows you to specify the progress bar in the launcher.

Progress bar in taskbar button:



To set the progress bar for a Window, you can use the BrowserWindow.setProgressBar API:

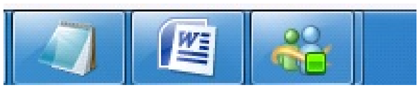
```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()
win.setProgressBar(0.5)
```

Icon Overlays in Taskbar (Windows)

On Windows a taskbar button can use a small overlay to display application status, as quoted from MSDN:

> Icon overlays serve as a contextual notification of status, and are intended to negate the need for a separate notification area status icon to communicate that information to the user. For instance, the new mail status in Microsoft Outlook, currently shown in the notification area, can now be indicated through an overlay on the taskbar button. Again, you must decide during your development cycle which method is best for your application. Overlay icons are intended to supply important, long-standing status or notifications such as network status, messenger status, or new mail. The user should not be presented with constantly changing overlays or animations.

Overlay on taskbar button:



To set the overlay icon for a window, you can use the BrowserWindow.setOverlayIcon API:

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()
win.setOverlayIcon('path/to/overlay.png', 'Description for overlay')
```

Flash Frame (Windows)

On Windows you can highlight the taskbar button to get the user's attention. This is similar to bouncing the dock icon on macOS. From the MSDN reference documentation:

> Typically, a window is flashed to inform the user that the window requires attention but that it does not currently have the keyboard focus.

To flash the BrowserWindow taskbar button, you can use the BrowserWindow.flashFrame API:

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()
win.once('focus', () => win.flashFrame(false))
win.flashFrame(true)
```

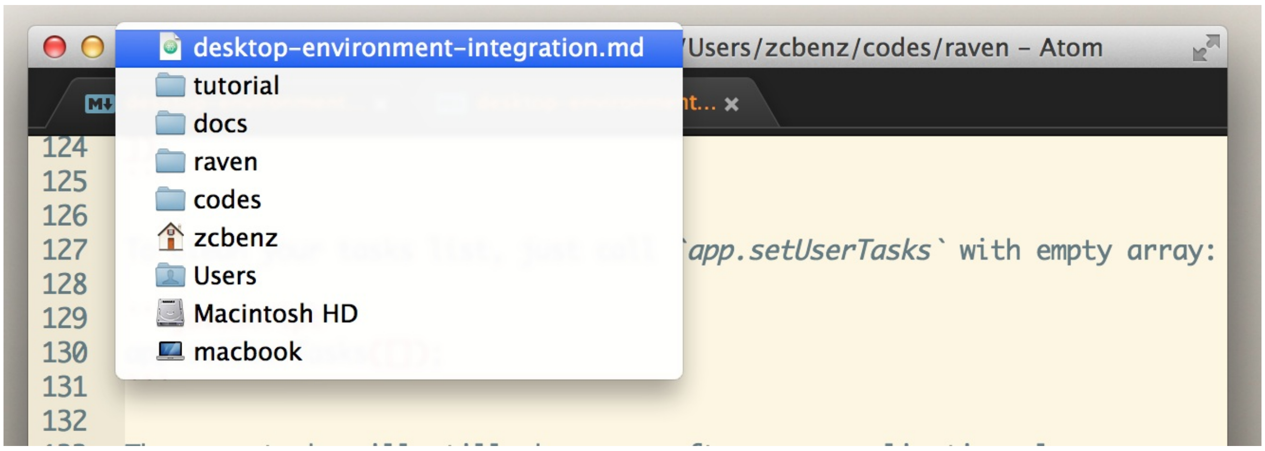
Don't forget to call the `flashFrame` method with `false` to turn off the flash. In the above example, it is called when the window comes into focus, but you might use a timeout or some other event to disable it.

Represented File of Window (macOS)

On macOS a window can set its represented file, so the file's icon can show in the title bar and when users Command-Click or Control-Click on the title a path popup will show.

You can also set the edited state of a window so that the file icon can indicate whether the document in this window has been modified.

Represented file popup menu:


desktop-environment-integration.md
tutorial
docs
raven
codes
zcbenz
Users
Macintosh HD
macbook
/Users/zcbenz/codes/raven – Atom
`app.setUserTasks` with empty array:

To set the represented file of window, you can use the BrowserWindow.setRepresentedFilename and BrowserWindow.setDocumentEdited APIs:

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()
win.setRepresentedFilename('/etc/passwd')
win.setDocumentEdited(true)
```

Dragging files out of the window

For certain kinds of apps that manipulate on files, it is important to be able to drag files from Electron to other apps. To implement this feature in your app, you need to call `webContents.startDrag(item)` API on `ondragstart` event.

In web page:

```
<a href="#" id="drag">item</a>
<script type="text/javascript" charset="utf-8">
  document.getElementById('drag').ondragstart = (event) => {
    event.preventDefault()
    ipcRenderer.send('ondragstart', '/path/to/item')
  }
</script>
```

In the main process:

```
const {ipcMain} = require('electron')
ipcMain.on('ondragstart', (event, filePath) => {
  event.sender.startDrag({
    file: filePath,
    icon: '/path/to/icon.png'
  })
})
```

Online/Offline Event Detection

Online and offline event detection can be implemented in the renderer process using standard HTML5 APIs, as shown in the following example.

main.js

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

let onlineStatusWindow

app.on('ready', () => {
  onlineStatusWindow = new BrowserWindow({ width: 0, height: 0, show: false })
  onlineStatusWindow.loadURL(`file://${__dirname}/online-status.html`)
})
```

online-status.html

```
<!DOCTYPE html>
<html>
<body>
<script>
  const alertOnlineStatus = () => {
    window.alert(navigator.onLine ? 'online' : 'offline')
  }

  window.addEventListener('online',  alertOnlineStatus)
  window.addEventListener('offline',  alertOnlineStatus)

  alertOnlineStatus()
</script>
</body>
</html>
```

There may be instances where you want to respond to these events in the main process as well. The main process however does not have a `navigator` object and thus cannot detect these events directly. Using Electron's inter-process communication utilities, the events can be forwarded to the main process and handled as needed, as shown in the following example.

main.js

```
const {app, BrowserWindow, ipcMain} = require('electron')
let onlineStatusWindow

app.on('ready', () => {
  onlineStatusWindow = new BrowserWindow({ width: 0, height: 0, show: false })
  onlineStatusWindow.loadURL(`file://${__dirname}/online-status.html`)
})

ipcMain.on('online-status-changed', (event, status) => {
  console.log(status)
})
```

online-status.html

```
<!DOCTYPE html>
<html>
<body>
<script>
  const {ipcRenderer} = require('electron')
  const updateOnlineStatus = () => {
    ipcRenderer.send('online-status-changed', navigator.onLine ? 'online' : 'offline')
  }

  window.addEventListener('online',  updateOnlineStatus)
  window.addEventListener('offline',  updateOnlineStatus)

  updateOnlineStatus()
</script>
</body>
</html>
```

NOTE: If Electron is not able to connect to a local area network (LAN) or a router, it is considered offline; all other conditions return `true`. So while you can assume that Electron is offline when `navigator.onLine` returns a `false` value, you cannot assume that a `true` value necessarily means that Electron can access the internet. You could be getting false positives, such as in cases where the computer is running a virtualization software that has virtual ethernet adapters that are always "connected." Therefore, if you really want to determine the internet access status of Electron, you should develop additional means for checking.

Synopsis

> How to use Node.js and Electron APIs.

All of Node.js's built-in modules are available in Electron and third-party node modules also fully supported as well (including the native modules).

Electron also provides some extra built-in modules for developing native desktop applications. Some modules are only available in the main process, some are only available in the renderer process (web page), and some can be used in both processes.

The basic rule is: if a module is GUI or low-level system related, then it should be only available in the main process. You need to be familiar with the concept of main process vs. renderer process scripts to be able to use those modules.

The main process script is just like a normal Node.js script:

```
const {app, BrowserWindow} = require('electron')
let win = null

app.on('ready', () => {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL('https://github.com')
})
```

The renderer process is no different than a normal web page, except for the extra ability to use node modules:

```
<!DOCTYPE html>
<html>
<body>
<script>
  const {app} = require('electron').remote
  console.log(app.getVersion())
</script>
</body>
</html>
```

To run your app, read Run your app.

Destructuring assignment

As of 0.37, you can use destructuring assignment to make it easier to use built-in modules.

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

let win

app.on('ready', () => {
  win = new BrowserWindow()
  win.loadURL('https://github.com')
})
```

If you need the entire `electron` module, you can require it and then using destructuring to access the individual modules from `electron`.

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

let win

app.on('ready', () => {
  win = new BrowserWindow()
  win.loadURL('https://github.com')
})
```

This is equivalent to the following code:

```
const electron = require('electron')
const app = electron.app
const BrowserWindow = electron.BrowserWindow
let win

app.on('ready', () => {
  win = new BrowserWindow()
  win.loadURL('https://github.com')
})
```

process

> Extensions to process object.

Process: Main, Renderer

Electron's `process` object is extended from the Node.js `process` object. It adds the following events, properties, and methods:

Events

Event: 'loaded'

Emitted when Electron has loaded its internal initialization script and is beginning to load the web page or the main script.

It can be used by the preload script to add removed Node global symbols back to the global scope when node integration is turned off:

```
// preload.js
const _setImmediate = setImmediate
const _clearImmediate = clearImmediate
process.once('loaded', () => {
  global.setImmediate = _setImmediate
  global.clearImmediate = _clearImmediate
})
```

Properties

`process.noAsar`

Setting this to `true` can disable the support for `asar` archives in Node's built-in modules.

`process.type`

Current process's type, can be `"browser"` (i.e. main process) or `"renderer"` .

`process.versions.electron`

Electron's version string.

`process.versions.chrome`

Chrome's version string.

`process.resourcesPath`

Path to the resources directory.

`process.mas`

For Mac App Store build, this property is `true` , for other builds it is `undefined` .

`process.windowsStore`

If the app is running as a Windows Store app (appx), this property is `true` , for otherwise it is `undefined` .

`process.defaultApp`

When app is started by being passed as parameter to the default app, this property is `true` in the main process, otherwise it is `undefined` .

Methods

The `process` object has the following method:

`process.crash()`

Causes the main thread of the current process crash.

`process.hang()`

Causes the main thread of the current process hang.

`process.setFdLimit(maxDescriptors)` *macOS Linux*

- `maxDescriptors` Integer

Sets the file descriptor soft limit to `maxDescriptors` or the OS hard limit, whichever is lower for the current process.

`process.getProcessMemoryInfo()`

Returns `Object` :

- `workingSetSize` Integer - The amount of memory currently pinned to actual physical RAM.
- `peakWorkingSetSize` Integer - The maximum amount of memory that has ever been pinned to actual physical RAM.
- `privateBytes` Integer - The amount of memory not shared by other processes, such as JS heap or HTML content.
- `sharedBytes` Integer - The amount of memory shared between processes, typically memory consumed by the Electron code itself

Returns an object giving memory usage statistics about the current process. Note that all statistics are reported in Kilobytes.

`process.getSystemMemoryInfo()`

Returns `Object` :

- `total` Integer - The total amount of physical memory in Kilobytes available to the system.
- `free` Integer - The total amount of memory not being used by applications or disk cache.
- `swapTotal` Integer - The total amount of swap memory in Kilobytes available to the system. *Windows Linux*
- `swapFree` Integer - The free amount of swap memory in Kilobytes available to the system. *Windows Linux*

Returns an object giving memory usage statistics about the entire system. Note that all statistics are reported in Kilobytes.

Supported Chrome Command Line Switches

> Command line switches supported by Electron.

You can use app.commandLine.appendSwitch to append them in your app's main script before the ready event of the app module is emitted:

```
const {app} = require('electron')
app.commandLine.appendSwitch('remote-debugging-port', '8315')
app.commandLine.appendSwitch('host-rules', 'MAP * 127.0.0.1')

app.on('ready', () => {
  // Your code here
})
```

--ignore-connections-limit= `domains`

Ignore the connections limit for `domains` list separated by `,` .

--disable-http-cache

Disables the disk cache for HTTP requests.

--disable-http2

Disable HTTP/2 and SPDY/3.1 protocols.

--debug= `port` and --debug-brk= `port`

Debug-related flags, see the Debugging the Main Process guide for details.

--remote-debugging-port= `port`

Enables remote debugging over HTTP on the specified `port` .

--js-flags= `flags`

Specifies the flags passed to the Node JS engine. It has to be passed when starting Electron if you want to enable the `flags` in the main process.

```
$ electron --js-flags="--harmony_proxies --harmony_collections" your-app
```

See the Node documentation or run `node --help` in your terminal for a list of available flags. Additionally, run `node --v8-options` to see a list of flags that specifically refer to Node's V8 JavaScript engine.

--proxy-server= `address:port`

Use a specified proxy server, which overrides the system setting. This switch only affects requests with HTTP protocol, including HTTPS and WebSocket requests. It is also noteworthy that not all proxy servers support HTTPS and WebSocket requests.

--proxy-bypass-list= `hosts`

Instructs Electron to bypass the proxy server for the given semi-colon-separated list of hosts. This flag has an effect only if used in tandem with `--proxy-server` .

For example:

```
const {app} = require('electron')
app.commandLine.appendSwitch('proxy-bypass-list', '<local>;*.google.com;*foo.com;1.2.3.4:5678')
```

Will use the proxy server for all hosts except for local addresses (`localhost` , `127.0.0.1` etc.), `google.com` subdomains, hosts that contain the suffix `foo.com` and anything at `1.2.3.4:5678` .

--proxy-pac-url= `url`

Uses the PAC script at the specified `url` .

--no-proxy-server

Don't use a proxy server and always make direct connections. Overrides any other proxy server flags that are passed.

--host-rules= `rules`

A comma-separated list of `rules` that control how hostnames are mapped.

For example:

- `MAP * 127.0.0.1` Forces all hostnames to be mapped to 127.0.0.1
- `MAP *.google.com proxy` Forces all google.com subdomains to be resolved to "proxy".
- `MAP test.com [::1]:77` Forces "test.com" to resolve to IPv6 loopback. Will also force the port of the resulting socket address to be 77.
- `MAP * baz, EXCLUDE www.google.com` Remaps everything to "baz", except for "www.google.com".

These mappings apply to the endpoint host in a net request (the TCP connect and host resolver in a direct connection, and the `CONNECT` in an HTTP proxy connection, and the endpoint host in a `SOCKS` proxy connection).

--host-resolver-rules= `rules`

Like `--host-rules` but these `rules` only apply to the host resolver.

--auth-server-whitelist= `url`

A comma-separated list of servers for which integrated authentication is enabled.

For example:

```
--auth-server-whitelist='*example.com, *foobar.com, *baz'
```

then any `url` ending with `example.com` , `foobar.com` , `baz` will be considered for integrated authentication. Without `*` prefix the url has to match exactly.

--auth-negotiate-delegate-whitelist= `url`

A comma-separated list of servers for which delegation of user credentials is required. Without `*` prefix the url has to match exactly.

--ignore-certificate-errors

Ignores certificate related errors.

--ppapi-flash-path= `path`

Sets the `path` of the pepper flash plugin.

--ppapi-flash-version= `version`

Sets the `version` of the pepper flash plugin.

--log-net-log= `path`

Enables net log events to be saved and writes them to `path` .

--disable-renderer-backgrounding

Prevents Chromium from lowering the priority of invisible pages' renderer processes.

This flag is global to all renderer processes, if you only want to disable throttling in one window, you can take the hack of playing silent audio.

--enable-logging

Prints Chromium's logging into console.

This switch can not be used in `app.commandLine.appendSwitch` since it is parsed earlier than user's app is loaded, but you can set the `ELECTRON_ENABLE_LOGGING` environment variable to achieve the same effect.

--v= `log_level`

Gives the default maximal active V-logging level; 0 is the default. Normally positive values are used for V-logging levels.

This switch only works when `--enable-logging` is also passed.

--vmodule= `pattern`

Gives the per-module maximal V-logging levels to override the value given by `--v` . E.g. `my_module=2,foo*=3` would change the logging level for all code in source files `my_module.*` and `foo*.*` .

Any pattern containing a forward or backward slash will be tested against the whole pathname and not just the module. E.g. `*/foo/bar/*=2` would change the logging level for all code in the source files under a `foo/bar` directory.

This switch only works when `--enable-logging` is also passed.

Environment Variables

> Control application configuration and behavior without changing code.

Certain Electron behaviors are controlled by environment variables because they are initialized earlier than the command line flags and the app's code.

POSIX shell example:

```
$ export ELECTRON_ENABLE_LOGGING=true
$ electron
```

Windows console example:

```
> set ELECTRON_ENABLE_LOGGING=true
> electron
```

Production Variables

The following environment variables are intended primarily for use at runtime in packaged Electron applications.

`GOOGLE_API_KEY`

Electron includes a hardcoded API key for making requests to Google's geocoding webservice. Because this API key is included in every version of Electron, it often exceeds its usage quota. To work around this, you can supply your own Google API key in the environment. Place the following code in your main process file, before opening any browser windows that will make geocoding requests:

```
process.env.GOOGLE_API_KEY = 'YOUR_KEY_HERE'
```

For instructions on how to acquire a Google API key, visit this page.

By default, a newly generated Google API key may not be allowed to make geocoding requests. To enable geocoding requests, visit this page.

`ELECTRON_NO_ASAR`

Disables ASAR support. This variable is only supported in forked child processes and spawned child processes that set `ELECTRON_RUN_AS_NODE`.

`ELECTRON_RUN_AS_NODE`

Starts the process as a normal Node.js process.

`ELECTRON_NO_ATTACH_CONSOLE` *Windows*

Don't attach to the current console session.

`ELECTRON_FORCE_WINDOW_MENU_BAR` *Linux*

Don't use the global menu bar on Linux.

Development Variables

The following environment variables are intended primarily for development and debugging purposes.

`ELECTRON_ENABLE_LOGGING`

Prints Chrome's internal logging to the console.

`ELECTRON_LOG_ASAR_READS`

When Electron reads from an ASAR file, log the read offset and file path to the system `tmpdir`. The resulting file can be provided to the ASAR module to optimize file ordering.

`ELECTRON_ENABLE_STACK_DUMPING`

Prints the stack trace to the console when Electron crashes.

This environment variable will not work if the `crashReporter` is started.

`ELECTRON_DEFAULT_ERROR_MODE` *Windows*

Shows the Windows's crash dialog when Electron crashes.

This environment variable will not work if the `crashReporter` is started.

`File` Object

> Use the HTML5 `File` API to work natively with files on the filesystem.

The DOM's File interface provides abstraction around native files in order to let users work on native files directly with the HTML5 file API. Electron has added a `path` attribute to the `File` interface which exposes the file's real path on filesystem.

Example of getting a real path from a dragged-onto-the-app file:

```
<div id="holder">
  Drag your file here
</div>

<script>
  const holder = document.getElementById('holder')
  holder.ondragover = () => {
    return false;
  }
  holder.ondragleave = holder.ondragend = () => {
    return false;
  }
  holder.ondrop = (e) => {
    e.preventDefault()
    for (let f of e.dataTransfer.files) {
      console.log('File(s) you dragged here: ', f.path)
    }
    return false;
  }
</script>
```

`<webview>` Tag

> Display external web content in an isolated frame and process.

Use the `webview` tag to embed 'guest' content (such as web pages) in your Electron app. The guest content is contained within the `webview` container. An embedded page within your app controls how the guest content is laid out and rendered.

Unlike an `iframe`, the `webview` runs in a separate process than your app. 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.

For security purposes, `webview` can only be used in `BrowserWindow`s that have `nodeIntegration` enabled.

Example

To embed a web page in your app, add the `webview` tag to your app's embedder page (this is the app page that will display the guest content). In its simplest form, the `webview` tag includes the `src` of the web page and css styles that control the appearance of the `webview` container:

```
<webview id="foo" src="https://www.github.com/" style="display:inline-flex; width:640px; height:480px"></webview>
```

If you want to control the guest content in any way, you can write JavaScript that listens for `webview` events and responds to those events using the `webview` methods. Here's sample code with two event listeners: one that listens for the web page to start loading, the other for the web page to stop loading, and displays a "loading..." message during the load time:

```
<script>
  onload = () => {
    const webview = document.getElementById('foo')
    const indicator = document.querySelector('.indicator')

    const loadstart = () => {
      indicator.innerText = 'loading...'
    }

    const loadstop = () => {
      indicator.innerText = ''
    }

    webview.addEventListener('did-start-loading', loadstart)
    webview.addEventListener('did-stop-loading', loadstop)
  }
</script>
```

CSS Styling Notes

Please note that the `webview` tag's style uses `display:flex;` internally to ensure the child `object` element fills the full height and width of its `webview` container when used with traditional and flexbox layouts (since v0.36.11). Please do not overwrite the default `display:flex;` CSS property, unless specifying `display:inline-flex;` for inline layout.

`webview` has issues being hidden using the `hidden` attribute or using `display: none;`. It can cause unusual rendering behaviour within its child `browserplugin` object and the web page is reloaded, when the `webview` is un-hidden, as opposed to just becoming visible again. The recommended approach is to hide the `webview` using CSS by zeroing the `width` & `height` and allowing the element to shrink to the 0px dimensions via `flex`.

```
<style>
  webview {
    display:inline-flex;
    width:640px;
    height:480px;
  }
  webview.hide {
    flex: 0 1;
    width: 0px;
    height: 0px;
  }
</style>
```

Tag Attributes

The `webview` tag has the following attributes:

`src`

```
<webview src="https://www.github.com/"></webview>
```

Returns the visible URL. Writing to this attribute initiates top-level navigation.

Assigning `src` its own value will reload the current page.

The `src` attribute can also accept data URLs, such as `data:text/plain,Hello, world!` .

`autosize`

```
<webview src="https://www.github.com/" autosize="on" minwidth="576" minheight="432"></webview>
```

If "on", the `webview` container will automatically resize within the bounds specified by the attributes `minwidth` , `minheight` , `maxwidth` , and `maxheight` . These constraints do not impact the `webview` unless `autosize` is enabled. When `autosize` is enabled, the `webview` container size cannot be less than the minimum values or greater than the maximum.

`nodeintegration`

```
<webview src="http://www.google.com/" nodeintegration></webview>
```

If "on", the guest page in `webview` will have node integration and can use node APIs like `require` and `process` to access low level system resources.

`plugins`

```
<webview src="https://www.github.com/" plugins></webview>
```

If "on", the guest page in `webview` will be able to use browser plugins.

`preload`

```
<webview src="https://www.github.com/" preload="./test.js"></webview>
```

Specifies a script that will be loaded before other scripts run in the guest page. The protocol of script's URL must be either `file:` or `asar:`, because it will be loaded by `require` in guest page under the hood.

When the guest page doesn't have node integration this script will still have access to all Node APIs, but global objects injected by Node will be deleted after this script has finished executing.

`httpreferrer`

```
<webview src="https://www.github.com/" httpreferrer="http://cheng.guru"></webview>
```

Sets the referrer URL for the guest page.

`useragent`

```
<webview src="https://www.github.com/" useragent="Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; AS; rv:11.0) like
Gecko"></webview>
```

Sets the user agent for the guest page before the page is navigated to. Once the page is loaded, use the `setUserAgent` method to change the user agent.

`disablewebsecurity`

```
<webview src="https://www.github.com/" disablewebsecurity></webview>
```

If "on", the guest page will have web security disabled.

`partition`

```
<webview src="https://github.com" partition="persist:github"></webview>
<webview src="http://electron.atom.io" partition="electron"></webview>
```

Sets the session used by the page. If `partition` starts with `persist:`, the page will use a persistent session available to all pages in the app with the same `partition`. if there is no `persist:` prefix, the page will use an in-memory session. By assigning the same `partition`, multiple pages can share the same session. If the `partition` is unset then default session of the app will be used.

This value can only be modified before the first navigation, since the session of an active renderer process cannot change. Subsequent attempts to modify the value will fail with a DOM exception.

`allowpopups`

```
<webview src="https://www.github.com/" allowpopups></webview>
```

If "on", the guest page will be allowed to open new windows.

`blinkfeatures`

```
<webview src="https://www.github.com/" blinkfeatures="PreciseMemoryInfo, CSSVariables"></webview>
```

A list of strings which specifies the blink features to be enabled separated by `,`. The full list of supported feature strings can be found in the RuntimeEnabledFeatures.in file.

`disableblinkfeatures`

```
<webview src="https://www.github.com/" disableblinkfeatures="PreciseMemoryInfo, CSSVariables"></webview>
```

A list of strings which specifies the blink features to be disabled separated by `,` . The full list of supported feature strings can be found in the RuntimeEnabledFeatures.in file.

Methods

The `webview` tag has the following methods:

Note: The webview element must be loaded before using the methods.

Example

```
const webview = document.getElementById('foo')
webview.addEventListener('dom-ready', () => {
  webview.openDevTools()
})
```

`<webview>.loadURL(url[, options])`

- `url` URL
- `options` Object (optional)
  - `httpReferrer` String - A HTTP Referrer url.
  - `userAgent` String - A user agent originating the request.
  - `extraHeaders` String - Extra headers separated by "\n"

Loads the `url` in the webview, the `url` must contain the protocol prefix, e.g. the `http://` or `file://` .

`<webview>.getURL()`

Returns URL of guest page.

`<webview>.getTitle()`

Returns the title of guest page.

`<webview>.isLoading()`

Returns a boolean whether guest page is still loading resources.

`<webview>.isWaitingForResponse()`

Returns a boolean whether the guest page is waiting for a first-response for the main resource of the page.

`<webview>.stop()`

Stops any pending navigation.

`<webview>.reload()`

Reloads the guest page.

`<webview>.reloadIgnoringCache()`

Reloads the guest page and ignores cache.

`<webview>.canGoBack()`

Returns a boolean whether the guest page can go back.

`<webview>.canGoForward()`

Returns a boolean whether the guest page can go forward.

`<webview>.canGoToOffset(offset)`

- `offset` Integer

Returns a boolean whether the guest page can go to `offset`.

`<webview>.clearHistory()`

Clears the navigation history.

`<webview>.goBack()`

Makes the guest page go back.

`<webview>.goForward()`

Makes the guest page go forward.

`<webview>.goToIndex(index)`

- `index` Integer

Navigates to the specified absolute index.

`<webview>.goToOffset(offset)`

- `offset` Integer

Navigates to the specified offset from the "current entry".

`<webview>.isCrashed()`

Whether the renderer process has crashed.

`<webview>.setUserAgent(userAgent)`

- `userAgent` String

Overrides the user agent for the guest page.

`<webview>.getUserAgent()`

Returns a `String` representing the user agent for guest page.

`<webview>.insertCSS(css)`

- `css` String

Injects CSS into the guest page.

`<webview>.executeJavaScript(code, userGesture, callback)`

- `code` String
- `userGesture` Boolean - Default `false` .
- `callback` Function (optional) - Called after script has been executed.
  - `result`

Evaluates `code` in page. If `userGesture` is set, it will create the user gesture context in the page. HTML APIs like `requestFullScreen` , which require user action, can take advantage of this option for automation.

`<webview>.openDevTools()`

Opens a DevTools window for guest page.

`<webview>.closeDevTools()`

Closes the DevTools window of guest page.

`<webview>.isDevToolsOpened()`

Returns a boolean whether guest page has a DevTools window attached.

`<webview>.isDevToolsFocused()`

Returns a boolean whether DevTools window of guest page is focused.

`<webview>.inspectElement(x, y)`

- `x` Integer
- `y` Integer

Starts inspecting element at position (`x` , `y`) of guest page.

`<webview>.inspectServiceWorker()`

Opens the DevTools for the service worker context present in the guest page.

`<webview>.setAudioMuted(muted)`

- `muted` Boolean

Set guest page muted.

`<webview>.isAudioMuted()`

Returns whether guest page has been muted.

`<webview>.undo()`

Executes editing command `undo` in page.

`<webview>.redo()`

Executes editing command `redo` in page.

`<webview>.cut()`

Executes editing command `cut` in page.

`<webview>.copy()`

Executes editing command `copy` in page.

`<webview>.paste()`

Executes editing command `paste` in page.

`<webview>.pasteAndMatchStyle()`

Executes editing command `pasteAndMatchStyle` in page.

`<webview>.delete()`

Executes editing command `delete` in page.

`<webview>.selectAll()`

Executes editing command `selectAll` in page.

`<webview>.unselect()`

Executes editing command `unselect` in page.

`<webview>.replace(text)`

- `text` String

Executes editing command `replace` in page.

`<webview>.replaceMisspelling(text)`

- `text` String

Executes editing command `replaceMisspelling` in page.

`<webview>.insertText(text)`

- `text` String

Inserts `text` to the focused element.

`<webview>.findInPage(text[, options])`

- `text` String - Content to be searched, must not be empty.
- `options` Object (optional)
  - `forward` Boolean - Whether to search forward or backward, defaults to `true` .
  - `findNext` Boolean - Whether the operation is first request or a follow up, defaults to `false` .

  - `matchCase` Boolean - Whether search should be case-sensitive, defaults to `false`.
  - `wordStart` Boolean - Whether to look only at the start of words. defaults to `false`.
  - `medialCapitalAsWordStart` Boolean - When combined with `wordStart`, accepts a match in the middle of a word if the match begins with an uppercase letter followed by a lowercase or non-letter. Accepts several other intra-word matches, defaults to `false`.

Starts a request to find all matches for the `text` in the web page and returns an `Integer` representing the request id used for the request. The result of the request can be obtained by subscribing to `found-in-page` event.

`<webview>.stopFindInPage(action)`

- `action` String - Specifies the action to take place when ending `<webview>.findInPage` request.
  - `clearSelection` - Clear the selection.
  - `keepSelection` - Translate the selection into a normal selection.
  - `activateSelection` - Focus and click the selection node.

Stops any `findInPage` request for the `webview` with the provided `action`.

`<webview>.print([options])`

Prints `webview`'s web page. Same as `webContents.print([options])`.

`<webview>.printToPDF(options, callback)`

Prints `webview`'s web page as PDF, Same as `webContents.printToPDF(options, callback)`.

`<webview>.capturePage([rect, ]callback)`

Captures a snapshot of the `webview`'s page. Same as `webContents.capturePage([rect, ]callback)`.

`<webview>.send(channel[, arg1][, arg2][, ...])`

- `channel` String
- `arg` (optional)

Send an asynchronous message to renderer process via `channel`, you can also send arbitrary arguments. The renderer process can handle the message by listening to the `channel` event with the `ipcRenderer` module.

See webContents.send for examples.

`<webview>.sendInputEvent(event)`

- `event` Object

Sends an input `event` to the page.

See webContents.sendInputEvent for detailed description of `event` object.

`<webview>.showDefinitionForSelection()` *macOS*

Shows pop-up dictionary that searches the selected word on the page.

`<webview>.getWebContents()`

Returns the WebContents associated with this `webview`.

DOM events

The following DOM events are available to the `webview` tag:

Event: 'load-commit'

Returns:

- `url` String
- `isMainFrame` Boolean

Fired when a load has committed. This includes navigation within the current document as well as subframe document-level loads, but does not include asynchronous resource loads.

Event: 'did-finish-load'

Fired when the navigation is done, i.e. the spinner of the tab will stop spinning, and the `onload` event is dispatched.

Event: 'did-fail-load'

Returns:

- `errorCode` Integer
- `errorDescription` String
- `validatedURL` String
- `isMainFrame` Boolean

This event is like `did-finish-load`, but fired when the load failed or was cancelled, e.g. `window.stop()` is invoked.

Event: 'did-frame-finish-load'

Returns:

- `isMainFrame` Boolean

Fired when a frame has done navigation.

Event: 'did-start-loading'

Corresponds to the points in time when the spinner of the tab starts spinning.

Event: 'did-stop-loading'

Corresponds to the points in time when the spinner of the tab stops spinning.

Event: 'did-get-response-details'

Returns:

- `status` Boolean
- `newURL` String
- `originalURL` String
- `httpResponseCode` Integer
- `requestMethod` String
- `referrer` String
- `headers` Object
- `resourceType` String

Fired when details regarding a requested resource is available. `status` indicates socket connection to download the resource.

Event: 'did-get-redirect-request'

Returns:

- `oldURL` String
- `newURL` String
- `isMainFrame` Boolean

Fired when a redirect was received while requesting a resource.

Event: 'dom-ready'

Fired when document in the given frame is loaded.

Event: 'page-title-updated'

Returns:

- `title` String
- `explicitSet` Boolean

Fired when page title is set during navigation. `explicitSet` is false when title is synthesized from file url.

Event: 'page-favicon-updated'

Returns:

- `favicons` Array - Array of URLs.

Fired when page receives favicon urls.

Event: 'enter-html-full-screen'

Fired when page enters fullscreen triggered by HTML API.

Event: 'leave-html-full-screen'

Fired when page leaves fullscreen triggered by HTML API.

Event: 'console-message'

Returns:

- `level` Integer
- `message` String
- `line` Integer
- `sourceId` String

Fired when the guest window logs a console message.

The following example code forwards all log messages to the embedder's console without regard for log level or other properties.

```
const webview = document.getElementById('foo')
webview.addEventListener('console-message', (e) => {
  console.log('Guest page logged a message:', e.message)
})
```

Event: 'found-in-page'

Returns:

- `result` Object
  - `requestId` Integer
  - `finalUpdate` Boolean - Indicates if more responses are to follow.
  - `activeMatchOrdinal` Integer (optional) - Position of the active match.
  - `matches` Integer (optional) - Number of Matches.
  - `selectionArea` Object (optional) - Coordinates of first match region.

Fired when a result is available for `webview.findInPage` request.

```
const webview = document.getElementById('foo')
webview.addEventListener('found-in-page', (e) => {
  if (e.result.finalUpdate) webview.stopFindInPage('keepSelection')
})

const requestId = webview.findInPage('test')
console.log(requestId)
```

Event: 'new-window'

Returns:

- `url` String
- `frameName` String
- `disposition` String - Can be `default` , `foreground-tab` , `background-tab` , `new-window` and `other` .
- `options` Object - The options which should be used for creating the new `BrowserWindow` .

Fired when the guest page attempts to open a new browser window.

The following example code opens the new url in system's default browser.

```
const {shell} = require('electron')
const webview = document.getElementById('foo')

webview.addEventListener('new-window', (e) => {
  const protocol = require('url').parse(e.url).protocol
  if (protocol === 'http:' || protocol === 'https:') {
    shell.openExternal(e.url)
  }
})
```

Event: 'will-navigate'

Returns:

- `url` String

Emitted when a user or the page wants to start navigation. It can happen when the `window.location` object is changed or a user clicks a link in the page.

This event will not emit when the navigation is started programmatically with APIs like `<webview>.loadURL` and `<webview>.back` .

It is also not emitted during in-page navigation, such as clicking anchor links or updating the `window.location.hash`. Use `did-navigate-in-page` event for this purpose.

Calling `event.preventDefault()` does **NOT** have any effect.

Event: 'did-navigate'

Returns:

- `url` String

Emitted when a navigation is done.

This event is not emitted for in-page navigations, such as clicking anchor links or updating the `window.location.hash`. Use `did-navigate-in-page` event for this purpose.

Event: 'did-navigate-in-page'

Returns:

- `url` String

Emitted when an in-page navigation happened.

When in-page navigation happens, the page URL changes but does not cause navigation outside of the page. Examples of this occurring are when anchor links are clicked or when the DOM `hashchange` event is triggered.

Event: 'close'

Fired when the guest page attempts to close itself.

The following example code navigates the `webview` to `about:blank` when the guest attempts to close itself.

```
const webview = document.getElementById('foo')
webview.addEventListener('close', () => {
  webview.src = 'about:blank'
})
```

Event: 'ipc-message'

Returns:

- `channel` String
- `args` Array

Fired when the guest page has sent an asynchronous message to embedder page.

With `sendToHost` method and `ipc-message` event you can easily communicate between guest page and embedder page:

```
// In embedder page.
const webview = document.getElementById('foo')
webview.addEventListener('ipc-message', (event) => {
  console.log(event.channel)
  // Prints "pong"
})
webview.send('ping')
```

```
// In guest page.
const {ipcRenderer} = require('electron')
ipcRenderer.on('ping', () => {
  ipcRenderer.sendToHost('pong')
})
```

Event: 'crashed'

Fired when the renderer process is crashed.

Event: 'gpu-crashed'

Fired when the gpu process is crashed.

Event: 'plugin-crashed'

Returns:

- `name` String
- `version` String

Fired when a plugin process is crashed.

Event: 'destroyed'

Fired when the WebContents is destroyed.

Event: 'media-started-playing'

Emitted when media starts playing.

Event: 'media-paused'

Emitted when media is paused or done playing.

Event: 'did-change-theme-color'

Returns:

- `themeColor` String

Emitted when a page's theme color changes. This is usually due to encountering a meta tag:

```
<meta name='theme-color' content='#ff0000'>
```

Event: 'update-target-url'

Returns:

- `url` String

Emitted when mouse moves over a link or the keyboard moves the focus to a link.

Event: 'devtools-opened'

Emitted when DevTools is opened.

Event: 'devtools-closed'

Emitted when DevTools is closed.

Event: 'devtools-focused'

Emitted when DevTools is focused / opened.

`window.open` Function

> Open a new window and load a URL.

When `window.open` is called to create a new window in a web page, a new instance of `BrowserWindow` will be created for the `url` and a proxy will be returned to `window.open` to let the page have limited control over it.

The proxy has limited standard functionality implemented to be compatible with traditional web pages. For full control of the new window you should create a `BrowserWindow` directly.

The newly created `BrowserWindow` will inherit the parent window's options by default. To override inherited options you can set them in the `features` string.

`window.open(url[, frameName][, features])`

- `url` String
- `frameName` String (optional)
- `features` String (optional)

Returns `BrowserWindowProxy` - Creates a new window and returns an instance of `BrowserWindowProxy` class.

The `features` string follows the format of standard browser, but each feature has to be a field of `BrowserWindow`'s options.

Notes:

- Node integration will always be disabled in the opened `window` if it is disabled on the parent window.
- Non-standard features (that are not handled by Chromium or Electron) given in `features` will be passed to any registered `webContent`'s `new-window` event handler in the `additionalFeatures` argument.

`window.opener.postMessage(message, targetOrigin)`

- `message` String
- `targetOrigin` String

Sends a message to the parent window with the specified origin or `*` for no origin preference.

app

> Control your application's event lifecycle.

Process: Main

The following example shows how to quit the application when the last window is closed:

```
const {app} = require('electron')
app.on('window-all-closed', () => {
  app.quit()
})
```

Events

The `app` object emits the following events:

Event: 'will-finish-launching'

Emitted when the application has finished basic startup. On Windows and Linux, the `will-finish-launching` event is the same as the `ready` event; on macOS, this event represents the `applicationWillFinishLaunching` notification of `NSApplication` . You would usually set up listeners for the `open-file` and `open-url` events here, and start the crash reporter and auto updater.

In most cases, you should just do everything in the `ready` event handler.

Event: 'ready'

Returns:

- `launchInfo` Object *macOS*

Emitted when Electron has finished initializing. On macOS, `launchInfo` holds the `userInfo` of the `NSUserNotification` that was used to open the application, if it was launched from Notification Center. You can call `app.isReady()` to check if this event has already fired.

Event: 'window-all-closed'

Emitted when all windows have been closed.

If you do not subscribe to this event and all windows are closed, the default behavior is to quit the app; however, if you subscribe, you control whether the app quits or not. If the user pressed `Cmd + Q` , or the developer called `app.quit()` , Electron will first try to close all the windows and then emit the `will-quit` event, and in this case the `window-all-closed` event would not be emitted.

Event: 'before-quit'

Returns:

- `event` Event

Emitted before the application starts closing its windows. Calling `event.preventDefault()` will prevent the default behaviour, which is terminating the application.

Note: If application quit was initiated by `autoUpdater.quitAndInstall()` then `before-quit` is emitted *after* emitting `close` event on all windows and closing them.

Event: 'will-quit'

Returns:

- `event` Event

Emitted when all windows have been closed and the application will quit. Calling `event.preventDefault()` will prevent the default behaviour, which is terminating the application.

See the description of the `window-all-closed` event for the differences between the `will-quit` and `window-all-closed` events.

Event: 'quit'

Returns:

- `event` Event
- `exitCode` Integer

Emitted when the application is quitting.

Event: 'open-file' *macOS*

Returns:

- `event` Event
- `path` String

Emitted when the user wants to open a file with the application. The `open-file` event is usually emitted when the application is already open and the OS wants to reuse the application to open the file. `open-file` is also emitted when a file is dropped onto the dock and the application is not yet running. Make sure to listen for the `open-file` event very early in your application startup to handle this case (even before the `ready` event is emitted).

You should call `event.preventDefault()` if you want to handle this event.

On Windows, you have to parse `process.argv` (in the main process) to get the filepath.

Event: 'open-url' *macOS*

Returns:

- `event` Event
- `url` String

Emitted when the user wants to open a URL with the application. Your application's `Info.plist` file must define the url scheme within the `CFBundleURLTypes` key, and set `NSPrincipalClass` to `AtomApplication` .

You should call `event.preventDefault()` if you want to handle this event.

Event: 'activate' *macOS*

Returns:

- `event` Event
- `hasVisibleWindows` Boolean

Emitted when the application is activated. Various actions can trigger this event, such as launching the application for the first time, attempting to re-launch the application when it's already running, or clicking on the application's dock or taskbar icon.

Event: 'continue-activity' *macOS*

Returns:

- `event` Event
- `type` String - A string identifying the activity. Maps to `NSUserActivity.activityType` .
- `userInfo` Object - Contains app-specific state stored by the activity on another device.

Emitted during Handoff when an activity from a different device wants to be resumed. You should call `event.preventDefault()` if you want to handle this event.

A user activity can be continued only in an app that has the same developer Team ID as the activity's source app and that supports the activity's type. Supported activity types are specified in the app's `Info.plist` under the `NSUserActivityTypes` key.

Event: 'browser-window-blur'

Returns:

- `event` Event
- `window` BrowserWindow

Emitted when a browserWindow gets blurred.

Event: 'browser-window-focus'

Returns:

- `event` Event
- `window` BrowserWindow

Emitted when a browserWindow gets focused.

Event: 'browser-window-created'

Returns:

- `event` Event
- `window` BrowserWindow

Emitted when a new browserWindow is created.

Event: 'web-contents-created'

Returns:

- `event` Event
- `webContents` WebContents

Emitted when a new webContents is created.

Event: 'certificate-error'

Returns:

- `event` Event
- `webContents` WebContents
- `url` String
- `error` String - The error code
- `certificate` Certificate
- `callback` Function
  - `isTrusted` Boolean - Whether to consider the certificate as trusted

Emitted when failed to verify the `certificate` for `url`, to trust the certificate you should prevent the default behavior with `event.preventDefault()` and call `callback(true)`.

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

app.on('certificate-error', (event, webContents, url, error, certificate, callback) => {
  if (url === 'https://github.com') {
    // Verification logic.
    event.preventDefault()
    callback(true)
  } else {
    callback(false)
  }
})
```

Event: 'select-client-certificate'

Returns:

- `event` Event
- `webContents` WebContents
- `url` URL
- `certificateList` Certificate[]
- `callback` Function
  - `certificate` Certificate (optional)

Emitted when a client certificate is requested.

The `url` corresponds to the navigation entry requesting the client certificate and `callback` can be called with an entry filtered from the list. Using `event.preventDefault()` prevents the application from using the first certificate from the store.

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

app.on('select-client-certificate', (event, webContents, url, list, callback) => {
  event.preventDefault()
  callback(list[0])
})
```

Event: 'login'

Returns:

- `event` Event
- `webContents` WebContents
- `request` Object
  - `method` String
  - `url` URL
  - `referrer` URL
- `authInfo` Object
  - `isProxy` Boolean
  - `scheme` String
  - `host` String
  - `port` Integer
  - `realm` String
- `callback` Function
  - `username` String
  - `password` String

Emitted when `webContents` wants to do basic auth.

The default behavior is to cancel all authentications, to override this you should prevent the default behavior with `event.preventDefault()` and call `callback(username, password)` with the credentials.

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

app.on('login', (event, webContents, request, authInfo, callback) => {
  event.preventDefault()
  callback('username', 'secret')
})
```

Event: 'gpu-process-crashed'

Returns:

- `event` Event
- `killed` Boolean

Emitted when the gpu process crashes or is killed.

Event: 'accessibility-support-changed' *macOS Windows*

Returns:

- `event` Event
- `accessibilitySupportEnabled` Boolean - `true` when Chrome's accessibility support is enabled, `false` otherwise.

Emitted when Chrome's accessibility support changes. This event fires when assistive technologies, such as screen readers, are enabled or disabled. See https://www.chromium.org/developers/design-documents/accessibility for more details.

Methods

The `app` object has the following methods:

Note: Some methods are only available on specific operating systems and are labeled as such.

`app.quit()`

Try to close all windows. The `before-quit` event will be emitted first. If all windows are successfully closed, the `will-quit` event will be emitted and by default the application will terminate.

This method guarantees that all `beforeunload` and `unload` event handlers are correctly executed. It is possible that a window cancels the quitting by returning `false` in the `beforeunload` event handler.

`app.exit([exitCode])`

- `exitCode` Integer (optional)

Exits immediately with `exitCode`. `exitCode` defaults to 0.

All windows will be closed immediately without asking user and the `before-quit` and `will-quit` events will not be emitted.

`app.relaunch([options])`

- `options` Object (optional)
  - `args` String[] - (optional)
  - `execPath` String (optional)

Relaunches the app when current instance exits.

By default the new instance will use the same working directory and command line arguments with current instance. When `args` is specified, the `args` will be passed as command line arguments instead. When `execPath` is specified, the `execPath` will be executed for relaunch instead of current app.

Note that this method does not quit the app when executed, you have to call `app.quit` or `app.exit` after calling `app.relaunch` to make the app restart.

When `app.relaunch` is called for multiple times, multiple instances will be started after current instance exited.

An example of restarting current instance immediately and adding a new command line argument to the new instance:

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

app.relaunch({args: process.argv.slice(1).concat(['--relaunch'])})
app.exit(0)
```

`app.isReady()`

Returns `Boolean` - `true` if Electron has finished initializing, `false` otherwise.

`app.focus()`

On Linux, focuses on the first visible window. On macOS, makes the application the active app. On Windows, focuses on the application's first window.

`app.hide()` *macOS*

Hides all application windows without minimizing them.

`app.show()` *macOS*

Shows application windows after they were hidden. Does not automatically focus them.

`app.getAppPath()`

Returns `String` - The current application directory.

`app.getPath(name)`

- `name` String

Returns `String` - A path to a special directory or file associated with `name`. On failure an `Error` is thrown.

You can request the following paths by the name:

- `home` User's home directory.
- `appData` Per-user application data directory, which by default points to:
  - `%APPDATA%` on Windows
  - `$XDG_CONFIG_HOME` or `~/.config` on Linux
  - `~/Library/Application Support` on macOS
- `userData` The directory for storing your app's configuration files, which by default it is the `appData` directory appended with your app's name.
- `temp` Temporary directory.
- `exe` The current executable file.
- `module` The `libchromiumcontent` library.
- `desktop` The current user's Desktop directory.

- `documents` Directory for a user's "My Documents".
- `downloads` Directory for a user's downloads.
- `music` Directory for a user's music.
- `pictures` Directory for a user's pictures.
- `videos` Directory for a user's videos.
- `pepperFlashSystemPlugin` Full path to the system version of the Pepper Flash plugin.

`app.getFileIcon(path[, options], callback)`

- `path` String
- `options` Object (optional)
  - `size` String
    - `small` - 16x16
    - `normal` - 32x32
    - `large` - 48x48 on *Linux*, 32x32 on *Windows*, unsupported on *macOS*.
- `callback` Function
  - `error` Error
  - `icon` NativeImage

Fetches a path's associated icon.

On *Windows*, there a 2 kinds of icons:

- Icons associated with certain file extensions, like `.mp3` , `.png` , etc.
- Icons inside the file itself, like `.exe` , `.dll` , `.ico` .

On *Linux* and *macOS*, icons depend on the application associated with file mime type.

`app.setPath(name, path)`

- `name` String
- `path` String

Overrides the `path` to a special directory or file associated with `name` . If the path specifies a directory that does not exist, the directory will be created by this method. On failure an `Error` is thrown.

You can only override paths of a `name` defined in `app.getPath` .

By default, web pages' cookies and caches will be stored under the `userData` directory. If you want to change this location, you have to override the `userData` path before the `ready` event of the `app` module is emitted.

`app.getVersion()`

Returns `String` - The version of the loaded application. If no version is found in the application's `package.json` file, the version of the current bundle or executable is returned.

`app.getName()`

Returns `String` - The current application's name, which is the name in the application's `package.json` file.

Usually the `name` field of `package.json` is a short lowercased name, according to the npm modules spec. You should usually also specify a `productName` field, which is your application's full capitalized name, and which will be preferred over `name` by Electron.

`app.setName(name)`

- `name` String

Overrides the current application's name.

`app.getLocale()`

Returns `String` - The current application locale. Possible return values are documented here.

Note: When distributing your packaged app, you have to also ship the `locales` folder.

Note: On Windows you have to call it after the `ready` events gets emitted.

`app.addRecentDocument(path)` *macOS Windows*

- `path` String

Adds `path` to the recent documents list.

This list is managed by the OS. On Windows you can visit the list from the task bar, and on macOS you can visit it from dock menu.

`app.clearRecentDocuments()` *macOS Windows*

Clears the recent documents list.

`app.setAsDefaultProtocolClient(protocol[, path, args])` *macOS Windows*

- `protocol` String - The name of your protocol, without `://`. If you want your app to handle `electron://` links, call this method with `electron` as the parameter.
- `path` String (optional) *Windows* - Defaults to `process.execPath`
- `args` String[] (optional) *Windows* - Defaults to an empty array

Returns `Boolean` - Whether the call succeeded.

This method sets the current executable as the default handler for a protocol (aka URI scheme). It allows you to integrate your app deeper into the operating system. Once registered, all links with `your-protocol://` will be opened with the current executable. The whole link, including protocol, will be passed to your application as a parameter.

On Windows you can provide optional parameters path, the path to your executable, and args, an array of arguments to be passed to your executable when it launches.

Note: On macOS, you can only register protocols that have been added to your app's `info.plist`, which can not be modified at runtime. You can however change the file with a simple text editor or script during build time. Please refer to Apple's documentation for details.

The API uses the Windows Registry and LSSetDefaultHandlerForURLScheme internally.

`app.removeAsDefaultProtocolClient(protocol[, path, args])` *macOS Windows*

- `protocol` String - The name of your protocol, without `://`.
- `path` String (optional) *Windows* - Defaults to `process.execPath`
- `args` String[] (optional) *Windows* - Defaults to an empty array

Returns `Boolean` - Whether the call succeeded.

This method checks if the current executable as the default handler for a protocol (aka URI scheme). If so, it will remove the app as the default handler.

`app.isDefaultProtocolClient(protocol[, path, args])` *macOS Windows*

- `protocol` String - The name of your protocol, without `://` .
- `path` String (optional) *Windows* - Defaults to `process.execPath`
- `args` String[] (optional) *Windows* - Defaults to an empty array

Returns `Boolean`

This method checks if the current executable is the default handler for a protocol (aka URI scheme). If so, it will return true. Otherwise, it will return false.

Note: On macOS, you can use this method to check if the app has been registered as the default protocol handler for a protocol. You can also verify this by checking `~/Library/Preferences/com.apple.LaunchServices.plist` on the macOS machine. Please refer to Apple's documentation for details.

The API uses the Windows Registry and LSCopyDefaultHandlerForURLScheme internally.

`app.setUserTasks(tasks)` *Windows*

- `tasks` Task[] - Array of `Task` objects

Adds `tasks` to the Tasks category of the JumpList on Windows.

`tasks` is an array of `Task` objects.

Returns `Boolean` - Whether the call succeeded.

Note: If you'd like to customize the Jump List even more use `app.setJumpList(categories)` instead.

`app.getJumpListSettings()` *Windows*

Returns `Object` :

- `minItems` Integer - The minimum number of items that will be shown in the Jump List (for a more detailed description of this value see the MSDN docs).
- `removedItems` JumpListItem[] - Array of `JumpListItem` objects that correspond to items that the user has explicitly removed from custom categories in the Jump List. These items must not be re-added to the Jump List in the **next** call to `app.setJumpList()` , Windows will not display any custom category that contains any of the removed items.

`app.setJumpList(categories)` *Windows*

- `categories` JumpListCategory[] or `null` - Array of `JumpListCategory` objects.

Sets or removes a custom Jump List for the application, and returns one of the following strings:

- `ok` - Nothing went wrong.
- `error` - One or more errors occurred, enable runtime logging to figure out the likely cause.
- `invalidSeparatorError` - An attempt was made to add a separator to a custom category in the Jump List. Separators are only allowed in the standard `Tasks` category.
- `fileTypeRegistrationError` - An attempt was made to add a file link to the Jump List for a file type the app isn't registered to handle.
- `customCategoryAccessDeniedError` - Custom categories can't be added to the Jump List due to user privacy or group policy settings.

If `categories` is `null` the previously set custom Jump List (if any) will be replaced by the standard Jump List for the app (managed by Windows).

Note: If a `JumpListCategory` object has neither the `type` nor the `name` property set then its `type` is assumed to be `tasks` . If the `name` property is set but the `type` property is omitted then the `type` is assumed to be `custom` .

Note: Users can remove items from custom categories, and Windows will not allow a removed item to be added back into a custom category until **after** the next successful call to `app.setJumpList(categories)`. Any attempt to re-add a removed item to a custom category earlier than that will result in the entire custom category being omitted from the Jump List. The list of removed items can be obtained using `app.getJumpListSettings()`.

Here's a very simple example of creating a custom Jump List:

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

app.setJumpList([
  {
    type: 'custom',
    name: 'Recent Projects',
    items: [
      { type: 'file', path: 'C:\\Projects\\project1.proj' },
      { type: 'file', path: 'C:\\Projects\\project2.proj' }
    ]
  },
  { // has a name so `type` is assumed to be "custom"
    name: 'Tools',
    items: [
      {
        type: 'task',
        title: 'Tool A',
        program: process.execPath,
        args: '--run-tool-a',
        icon: process.execPath,
        iconIndex: 0,
        description: 'Runs Tool A'
      },
      {
        type: 'task',
        title: 'Tool B',
        program: process.execPath,
        args: '--run-tool-b',
        icon: process.execPath,
        iconIndex: 0,
        description: 'Runs Tool B'
      }
    ]
  },
  { type: 'frequent' },
  { // has no name and no type so `type` is assumed to be "tasks"
    items: [
      {
        type: 'task',
        title: 'New Project',
        program: process.execPath,
        args: '--new-project',
        description: 'Create a new project.'
      },
      { type: 'separator' },
      {
        type: 'task',
        title: 'Recover Project',
        program: process.execPath,
        args: '--recover-project',
        description: 'Recover Project'
      }
    ]
  }
])
```

`app.makeSingleInstance(callback)`

- `callback` Function
  - `argv` String[] - An array of the second instance's command line arguments

  - `workingDirectory` String - The second instance's working directory

This method makes your application a Single Instance Application - instead of allowing multiple instances of your app to run, this will ensure that only a single instance of your app is running, and other instances signal this instance and exit.

`callback` will be called with `callback(argv, workingDirectory)` when a second instance has been executed. `argv` is an Array of the second instance's command line arguments, and `workingDirectory` is its current working directory. Usually applications respond to this by making their primary window focused and non-minimized.

The `callback` is guaranteed to be executed after the `ready` event of `app` gets emitted.

This method returns `false` if your process is the primary instance of the application and your app should continue loading. And returns `true` if your process has sent its parameters to another instance, and you should immediately quit.

On macOS the system enforces single instance automatically when users try to open a second instance of your app in Finder, and the `open-file` and `open-url` events will be emitted for that. However when users start your app in command line the system's single instance mechanism will be bypassed and you have to use this method to ensure single instance.

An example of activating the window of primary instance when a second instance starts:

```
const {app} = require('electron')
let myWindow = null

const shouldQuit = app.makeSingleInstance((commandLine, workingDirectory) => {
  // Someone tried to run a second instance, we should focus our window.
  if (myWindow) {
    if (myWindow.isMinimized()) myWindow.restore()
    myWindow.focus()
  }
})

if (shouldQuit) {
  app.quit()
}

// Create myWindow, load the rest of the app, etc...
app.on('ready', () => {
})
```

`app.releaseSingleInstance()`

Releases all locks that were created by `makeSingleInstance`. This will allow multiple instances of the application to once again run side by side.

`app.setUserActivity(type, userInfo[, webpageURL])` *macOS*

- `type` String - Uniquely identifies the activity. Maps to `NSUserActivity.activityType`.
- `userInfo` Object - App-specific state to store for use by another device.
- `webpageURL` String (optional) - The webpage to load in a browser if no suitable app is installed on the resuming device. The scheme must be `http` or `https`.

Creates an `NSUserActivity` and sets it as the current activity. The activity is eligible for Handoff to another device afterward.

`app.getCurrentActivityType()` *macOS*

Returns `String` - The type of the currently running activity.

`app.setAppUserModelId(id)` *Windows*

- `id` String

Changes the Application User Model ID to `id`.

`app.importCertificate(options, callback)` *LINUX*

- `options` Object
  - `certificate` String - Path for the pkcs12 file.
  - `password` String - Passphrase for the certificate.
- `callback` Function
  - `result` Integer - Result of import.

Imports the certificate in pkcs12 format into the platform certificate store. `callback` is called with the `result` of import operation, a value of `0` indicates success while any other value indicates failure according to chromium net_error_list.

`app.disableHardwareAcceleration()`

Disables hardware acceleration for current app.

This method can only be called before app is ready.

`app.setBadgeCount(count)` *Linux macOS*

- `count` Integer

Returns `Boolean` - Whether the call succeeded.

Sets the counter badge for current app. Setting the count to `0` will hide the badge.

On macOS it shows on the dock icon. On Linux it only works for Unity launcher,

Note: Unity launcher requires the existence of a `.desktop` file to work, for more information please read Desktop Environment Integration.

`app.getBadgeCount()` *Linux macOS*

Returns `Integer` - The current value displayed in the counter badge.

`app.isUnityRunning()` *Linux*

Returns `Boolean` - Whether the current desktop environment is Unity launcher.

`app.getLoginItemSettings([options])` *macOS Windows*

- `options` Object (optional)
  - `path` String (optional) *Windows* - The executable path to compare against. Defaults to `process.execPath`.
  - `args` String[] (optional) *Windows* - The command-line arguments to compare against. Defaults to an empty array.

If you provided `path` and `args` options to `app.setLoginItemSettings` then you need to pass the same arguments here for `openAtLogin` to be set correctly.

Returns `Object`:

- `openAtLogin` Boolean - `true` if the app is set to open at login.
- `openAsHidden` Boolean - `true` if the app is set to open as hidden at login. This setting is only supported on macOS.
- `wasOpenedAtLogin` Boolean - `true` if the app was opened at login automatically. This setting is only supported on macOS.
- `wasOpenedAsHidden` Boolean - `true` if the app was opened as a hidden login item. This indicates that the app should not open any windows at startup. This setting is only supported on macOS.
- `restoreState` Boolean - `true` if the app was opened as a login item that should restore the state from the previous

session. This indicates that the app should restore the windows that were open the last time the app was closed. This setting is only supported on macOS.

Note: This API has no effect on MAS builds.

`app.setLoginItemSettings(settings[, path, args])` *macOS Windows*

- `settings` Object
  - `openAtLogin` Boolean (optional) - `true` to open the app at login, `false` to remove the app as a login item. Defaults to `false`.
  - `openAsHidden` Boolean (optional) - `true` to open the app as hidden. Defaults to `false`. The user can edit this setting from the System Preferences so `app.getLoginItemStatus().wasOpenedAsHidden` should be checked when the app is opened to know the current value. This setting is only supported on macOS.
- `path` String (optional) *Windows* - The executable to launch at login. Defaults to `process.execPath`.
- `args` String[] (optional) *Windows* - The command-line arguments to pass to the executable. Defaults to an empty array. Take care to wrap paths in quotes.

Set the app's login item settings.

To work with Electron's `autoUpdater` on Windows, which uses Squirrel, you'll want to set the launch path to Update.exe, and pass arguments that specify your application name. For example:

```
const appFolder = path.dirname(process.execPath)
const updateExe = path.resolve(appFolder, '..', 'Update.exe')
const exeName = path.basename(process.execPath)

app.setLoginItemSettings({
  openAtLogin: true,
  path: updateExe,
  args: [
    '--processStart', `"${exeName}"`,
    '--process-start-args', `"--hidden"`
  ]
})
```

Note: This API has no effect on MAS builds.

`app.isAccessibilitySupportEnabled()` *macOS Windows*

Returns `Boolean` - `true` if Chrome's accessibility support is enabled, `false` otherwise. This API will return `true` if the use of assistive technologies, such as screen readers, has been detected. See https://www.chromium.org/developers/design-documents/accessibility for more details.

`app.setAboutPanelOptions(options)` *macOS*

- `options` Object
  - `applicationName` String (optional) - The app's name.
  - `applicationVersion` String (optional) - The app's version.
  - `copyright` String (optional) - Copyright information.
  - `credits` String (optional) - Credit information.
  - `version` String (optional) - The app's build version number.

Set the about panel options. This will override the values defined in the app's `.plist` file. See the Apple docs for more details.

`app.commandLine.appendSwitch(switch[, value])`

- `switch` String - A command-line switch

- `value` String (optional) - A value for the given switch

Append a switch (with optional `value`) to Chromium's command line.

Note: This will not affect `process.argv` , and is mainly used by developers to control some low-level Chromium behaviors.

`app.commandLine.appendArgument(value)`

- `value` String - The argument to append to the command line

Append an argument to Chromium's command line. The argument will be quoted correctly.

Note: This will not affect `process.argv` .

`app.dock.bounce([type])` *macOS*

- `type` String (optional) - Can be `critical` or `informational` . The default is `informational`

When `critical` is passed, the dock icon will bounce until either the application becomes active or the request is canceled.

When `informational` is passed, the dock icon will bounce for one second. However, the request remains active until either the application becomes active or the request is canceled.

Returns `Integer` an ID representing the request.

`app.dock.cancelBounce(id)` *macOS*

- `id` Integer

Cancel the bounce of `id` .

`app.dock.downloadFinished(filePath)` *macOS*

- `filePath` String

Bounces the Downloads stack if the filePath is inside the Downloads folder.

`app.dock.setBadge(text)` *macOS*

- `text` String

Sets the string to be displayed in the dock's badging area.

`app.dock.getBadge()` *macOS*

Returns `String` - The badge string of the dock.

`app.dock.hide()` *macOS*

Hides the dock icon.

`app.dock.show()` *macOS*

Shows the dock icon.

`app.dock.isVisible()` *macOS*

Returns `Boolean` - Whether the dock icon is visible. The `app.dock.show()` call is asynchronous so this method might not return true immediately after that call.

`app.dock.setMenu(menu)` *macOS*

- `menu` Menu

Sets the application's dock menu.

`app.dock.setIcon(image)` *macOS*

- `image` (NativeImage | String)

Sets the `image` associated with this dock icon.

autoUpdater

> Enable apps to automatically update themselves.

Process: Main

The `autoUpdater` module provides an interface for the Squirrel framework.

You can quickly launch a multi-platform release server for distributing your application by using one of these projects:

- nuts: *A smart release server for your applications, using GitHub as a backend. Auto-updates with Squirrel (Mac & Windows)*
- electron-release-server: *A fully featured, self-hosted release server for electron applications, compatible with auto-updater*
- squirrel-updates-server: *A simple node.js server for Squirrel.Mac and Squirrel.Windows which uses GitHub releases*
- squirrel-release-server: *A simple PHP application for Squirrel.Windows which reads updates from a folder. Supports delta updates.*

Platform notices

Though `autoUpdater` provides a uniform API for different platforms, there are still some subtle differences on each platform.

macOS

On macOS, the `autoUpdater` module is built upon Squirrel.Mac, meaning you don't need any special setup to make it work. For server-side requirements, you can read Server Support. Note that App Transport Security (ATS) applies to all requests made as part of the update process. Apps that need to disable ATS can add the `NSAllowsArbitraryLoads` key to their app's plist.

Note: Your application must be signed for automatic updates on macOS. This is a requirement of `Squirrel.Mac`.

Windows

On Windows, you have to install your app into a user's machine before you can use the `autoUpdater`, so it is recommended that you use the electron-winstaller, electron-builder or the grunt-electron-installer package to generate a Windows installer.

When using electron-winstaller or electron-builder make sure you do not try to update your app the first time it runs (Also see this issue for more info). It's also recommended to use electron-squirrel-startup to get desktop shortcuts for your app.

The installer generated with Squirrel will create a shortcut icon with an Application User Model ID in the format of `com.squirrel.PACKAGE_ID.YOUR_EXE_WITHOUT_DOT_EXE`, examples are `com.squirrel.slack.Slack` and `com.squirrel.code.Code`. You have to use the same ID for your app with `app.setAppUserModelId` API, otherwise Windows will not be able to pin your app properly in task bar.

The server-side setup is also different from macOS. You can read the documents of Squirrel.Windows to get more details.

Linux

There is no built-in support for auto-updater on Linux, so it is recommended to use the distribution's package manager to update your app.

Events

The `autoUpdater` object emits the following events:

Event: 'error'

Returns:

- `error` Error

Emitted when there is an error while updating.

Event: 'checking-for-update'

Emitted when checking if an update has started.

Event: 'update-available'

Emitted when there is an available update. The update is downloaded automatically.

Event: 'update-not-available'

Emitted when there is no available update.

Event: 'update-downloaded'

Returns:

- `event` Event
- `releaseNotes` String
- `releaseName` String
- `releaseDate` Date
- `updateURL` String

Emitted when an update has been downloaded.

On Windows only `releaseName` is available.

Methods

The `autoUpdater` object has the following methods:

`autoUpdater.setFeedURL(url[, requestHeaders])`

- `url` String
- `requestHeaders` Object *macOS* (optional) - HTTP request headers.

Sets the `url` and initialize the auto updater.

`autoUpdater.getFeedURL()`

Returns `String` - The current update feed URL.

`autoUpdater.checkForUpdates()`

Asks the server whether there is an update. You must call `setFeedURL` before using this API.

`autoUpdater.quitAndInstall()`

Restarts the app and installs the update after it has been downloaded. It should only be called after `update-downloaded` has been emitted.

Note: `autoUpdater.quitAndInstall()` will close all application windows first and only emit `before-quit` event on `app` after that. This is different from the normal quit event sequence.

BrowserWindow

> Create and control browser windows.

Process: Main

```
// In the main process.
const {BrowserWindow} = require('electron')

// Or use `remote` from the renderer process.
// const {BrowserWindow} = require('electron').remote

let win = new BrowserWindow({width: 800, height: 600})
win.on('closed', () => {
  win = null
})

// Load a remote URL
win.loadURL('https://github.com')

// Or load a local HTML file
win.loadURL(`file://${__dirname}/app/index.html`)
```

Frameless window

To create a window without chrome, or a transparent window in arbitrary shape, you can use the Frameless Window API.

Showing window gracefully

When loading a page in the window directly, users may see the page load incrementally, which is not a good experience for a native app. To make the window display without visual flash, there are two solutions for different situations.

Using `ready-to-show` event

While loading the page, the `ready-to-show` event will be emitted when renderer process has done drawing for the first time, showing window after this event will have no visual flash:

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow({show: false})
win.once('ready-to-show', () => {
  win.show()
})
```

This event is usually emitted after the `did-finish-load` event, but for pages with many remote resources, it may be emitted before the `did-finish-load` event.

Setting `backgroundColor`

For a complex app, the `ready-to-show` event could be emitted too late, making the app feel slow. In this case, it is recommended to show the window immediately, and use a `backgroundColor` close to your app's background:

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

let win = new BrowserWindow({backgroundColor: '#2e2c29'})
win.loadURL('https://github.com')
```

Note that even for apps that use `ready-to-show` event, it is still recommended to set `backgroundColor` to make app feel more native.

Parent and child windows

By using `parent` option, you can create child windows:

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

let top = new BrowserWindow()
let child = new BrowserWindow({parent: top})
child.show()
top.show()
```

The `child` window will always show on top of the `top` window.

Modal windows

A modal window is a child window that disables parent window, to create a modal window, you have to set both `parent` and `modal` options:

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

let child = new BrowserWindow({parent: top, modal: true, show: false})
child.loadURL('https://github.com')
child.once('ready-to-show', () => {
  child.show()
})
```

Platform notices

- On macOS modal windows will be displayed as sheets attached to the parent window.
- On macOS the child windows will keep the relative position to parent window when parent window moves, while on Windows and Linux child windows will not move.
- On Windows it is not supported to change parent window dynamically.
- On Linux the type of modal windows will be changed to `dialog`.
- On Linux many desktop environments do not support hiding a modal window.

Class: BrowserWindow

> Create and control browser windows.

Process: Main

`BrowserWindow` is an EventEmitter.

It creates a new `BrowserWindow` with native properties as set by the `options`.

`new BrowserWindow([options])`

- `options` Object (optional)
  - `width` Integer (optional) - Window's width in pixels. Default is `800`.
  - `height` Integer (optional) - Window's height in pixels. Default is `600`.
  - `x` Integer (optional) (**required** if y is used) - Window's left offset from screen. Default is to center the window.
  - `y` Integer (optional) (**required** if x is used) - Window's top offset from screen. Default is to center the window.
  - `useContentSize` Boolean (optional) - The `width` and `height` would be used as web page's size, which means

the actual window's size will include window frame's size and be slightly larger. Default is `false` .

- `center` Boolean (optional) - Show window in the center of the screen.
- `minWidth` Integer (optional) - Window's minimum width. Default is `0` .
- `minHeight` Integer (optional) - Window's minimum height. Default is `0` .
- `maxWidth` Integer (optional) - Window's maximum width. Default is no limit.
- `maxHeight` Integer (optional) - Window's maximum height. Default is no limit.
- `resizable` Boolean (optional) - Whether window is resizable. Default is `true` .
- `movable` Boolean (optional) - Whether window is movable. This is not implemented on Linux. Default is `true` .
- `minimizable` Boolean (optional) - Whether window is minimizable. This is not implemented on Linux. Default is `true` .
- `maximizable` Boolean (optional) - Whether window is maximizable. This is not implemented on Linux. Default is `true` .
- `closable` Boolean (optional) - Whether window is closable. This is not implemented on Linux. Default is `true` .
- `focusable` Boolean (optional) - Whether the window can be focused. Default is `true` . On Windows setting `focusable: false` also implies setting `skipTaskbar: true` . On Linux setting `focusable: false` makes the window stop interacting with wm, so the window will always stay on top in all workspaces.
- `alwaysOnTop` Boolean (optional) - Whether the window should always stay on top of other windows. Default is `false` .
- `fullscreen` Boolean (optional) - Whether the window should show in fullscreen. When explicitly set to `false` the fullscreen button will be hidden or disabled on macOS. Default is `false` .
- `fullscreenable` Boolean (optional) - Whether the window can be put into fullscreen mode. On macOS, also whether the maximize/zoom button should toggle full screen mode or maximize window. Default is `true` .
- `skipTaskbar` Boolean (optional) - Whether to show the window in taskbar. Default is `false` .
- `kiosk` Boolean (optional) - The kiosk mode. Default is `false` .
- `title` String (optional) - Default window title. Default is `"Electron"` .
- `icon` (NativeImage | String) (optional) - The window icon. On Windows it is recommended to use `ICO` icons to get best visual effects, you can also leave it undefined so the executable's icon will be used.
- `show` Boolean (optional) - Whether window should be shown when created. Default is `true` .
- `frame` Boolean (optional) - Specify `false` to create a Frameless Window. Default is `true` .
- `parent` BrowserWindow (optional) - Specify parent window. Default is `null` .
- `modal` Boolean (optional) - Whether this is a modal window. This only works when the window is a child window. Default is `false` .
- `acceptFirstMouse` Boolean (optional) - Whether the web view accepts a single mouse-down event that simultaneously activates the window. Default is `false` .
- `disableAutoHideCursor` Boolean (optional) - Whether to hide cursor when typing. Default is `false` .
- `autoHideMenuBar` Boolean (optional) - Auto hide the menu bar unless the `Alt` key is pressed. Default is `false` .
- `enableLargerThanScreen` Boolean (optional) - Enable the window to be resized larger than screen. Default is `false` .
- `backgroundColor` String (optional) - Window's background color as Hexadecimal value, like `#66CD00` or `#FFF` or `#80FFFFFF` (alpha is supported). Default is `#FFF` (white).
- `hasShadow` Boolean (optional) - Whether window should have a shadow. This is only implemented on macOS. Default is `true` .
- `darkTheme` Boolean (optional) - Forces using dark theme for the window, only works on some GTK+3 desktop environments. Default is `false` .
- `transparent` Boolean (optional) - Makes the window transparent. Default is `false` .
- `type` String (optional) - The type of window, default is normal window. See more about this below.
- `titleBarStyle` String (optional) - The style of window title bar. Default is `default` . Possible values are:
  - `default` - Results in the standard gray opaque Mac title bar.
  - `hidden` - Results in a hidden title bar and a full size content window, yet the title bar still has the standard window controls ("traffic lights") in the top left.
  - `hidden-inset` - Results in a hidden title bar with an alternative look where the traffic light buttons are slightly more inset from the window edge.
- `thickFrame` Boolean (optional) - Use `WS_THICKFRAME` style for frameless windows on Windows, which adds

standard window frame. Setting it to `false` will remove window shadow and window animations. Default is `true`.
- `vibrancy` String (optional) - Add a type of vibrancy effect to the window, only on macOS. Can be `appearance-based`, `light`, `dark`, `titlebar`, `selection`, `menu`, `popover`, `sidebar`, `medium-light` or `ultra-dark`.
- `zoomToPageWidth` Boolean (optional) - Controls the behavior on macOS when option-clicking the green stoplight button on the toolbar or by clicking the Window > Zoom menu item. If `true`, the window will grow to the preferred width of the web page when zoomed, `false` will cause it to zoom to the width of the screen. This will also affect the behavior when calling `maximize()` directly. Default is `false`.
- `webPreferences` Object (optional) - Settings of web page's features.
  - `devTools` Boolean (optional) - Whether to enable DevTools. If it is set to `false`, can not use `BrowserWindow.webContents.openDevTools()` to open DevTools. Default is `true`.
  - `nodeIntegration` Boolean (optional) - Whether node integration is enabled. Default is `true`.
  - `preload` String (optional) - Specifies a script that will be loaded before other scripts run in the page. This script will always have access to node APIs no matter whether node integration is turned on or off. The value should be the absolute file path to the script. When node integration is turned off, the preload script can reintroduce Node global symbols back to the global scope. See example here.
  - `session` Session (optional) - Sets the session used by the page. Instead of passing the Session object directly, you can also choose to use the `partition` option instead, which accepts a partition string. When both `session` and `partition` are provided, `session` will be preferred. Default is the default session.
  - `partition` String (optional) - Sets the session used by the page according to the session's partition string. If `partition` starts with `persist:`, the page will use a persistent session available to all pages in the app with the same `partition`. If there is no `persist:` prefix, the page will use an in-memory session. By assigning the same `partition`, multiple pages can share the same session. Default is the default session.
  - `zoomFactor` Number (optional) - The default zoom factor of the page, `3.0` represents `300%`. Default is `1.0`.
  - `javascript` Boolean (optional) - Enables JavaScript support. Default is `true`.
  - `webSecurity` Boolean (optional) - When `false`, it will disable the same-origin policy (usually using testing websites by people), and set `allowRunningInsecureContent` to `true` if this options has not been set by user. Default is `true`.
  - `allowRunningInsecureContent` Boolean (optional) - Allow an https page to run JavaScript, CSS or plugins from http URLs. Default is `false`.
  - `images` Boolean (optional) - Enables image support. Default is `true`.
  - `textAreasAreResizable` Boolean (optional) - Make TextArea elements resizable. Default is `true`.
  - `webgl` Boolean (optional) - Enables WebGL support. Default is `true`.
  - `webaudio` Boolean (optional) - Enables WebAudio support. Default is `true`.
  - `plugins` Boolean (optional) - Whether plugins should be enabled. Default is `false`.
  - `experimentalFeatures` Boolean (optional) - Enables Chromium's experimental features. Default is `false`.
  - `experimentalCanvasFeatures` Boolean (optional) - Enables Chromium's experimental canvas features. Default is `false`.
  - `scrollBounce` Boolean (optional) - Enables scroll bounce (rubber banding) effect on macOS. Default is `false`.
  - `blinkFeatures` String (optional) - A list of feature strings separated by `,`, like `CSSVariables,KeyboardEventKey` to enable. The full list of supported feature strings can be found in the RuntimeEnabledFeatures.json5 file.
  - `disableBlinkFeatures` String (optional) - A list of feature strings separated by `,`, like `CSSVariables,KeyboardEventKey` to disable. The full list of supported feature strings can be found in the RuntimeEnabledFeatures.json5 file.
  - `defaultFontFamily` Object (optional) - Sets the default font for the font-family.
    - `standard` String (optional) - Defaults to `Times New Roman`.
    - `serif` String (optional) - Defaults to `Times New Roman`.
    - `sansSerif` String (optional) - Defaults to `Arial`.
    - `monospace` String (optional) - Defaults to `Courier New`.
    - `cursive` String (optional) - Defaults to `Script`.
    - `fantasy` String (optional) - Defaults to `Impact`.

    - `defaultFontSize` Integer (optional) - Defaults to `16` .
    - `defaultMonospaceFontSize` Integer (optional) - Defaults to `13` .
    - `minimumFontSize` Integer (optional) - Defaults to `0` .
    - `defaultEncoding` String (optional) - Defaults to `ISO-8859-1` .
    - `backgroundThrottling` Boolean (optional) - Whether to throttle animations and timers when the page becomes background. Defaults to `true` .
    - `offscreen` Boolean (optional) - Whether to enable offscreen rendering for the browser window. Defaults to `false` . See the offscreen rendering tutorial for more details.
    - `sandbox` Boolean (optional) - Whether to enable Chromium OS-level sandbox.
    - `contextIsolation` Boolean (optional) - Whether to run Electron APIs and the specified `preload` script in a separate JavaScript context. Defaults to `false` . The context that the `preload` script runs in will still have full access to the `document` and `window` globals but it will use its own set of JavaScript builtins (`Array` , `Object` , `JSON` , etc.) and will be isolated from any changes made to the global environment by the loaded page. The Electron API will only be available in the `preload` script and not the loaded page. This option should be used when loading potentially untrusted remote content to ensure the loaded content cannot tamper with the `preload` script and any Electron APIs being used. This option uses the same technique used by Chrome Content Scripts. You can access this context in the dev tools by selecting the 'Electron Isolated Context' entry in the combo box at the top of the Console tab. **Note:** This option is currently experimental and may change or be removed in future Electron releases.

When setting minimum or maximum window size with `minWidth` / `maxWidth` / `minHeight` / `maxHeight` , it only constrains the users. It won't prevent you from passing a size that does not follow size constraints to `setBounds` / `setSize` or to the constructor of `BrowserWindow` .

The possible values and behaviors of the `type` option are platform dependent. Possible values are:

- On Linux, possible types are `desktop` , `dock` , `toolbar` , `splash` , `notification` .
- On macOS, possible types are `desktop` , `textured` .
    - The `textured` type adds metal gradient appearance (`NSTexturedBackgroundWindowMask`).
    - The `desktop` type places the window at the desktop background window level (`kCGDesktopWindowLevel - 1`). Note that desktop window will not receive focus, keyboard or mouse events, but you can use `globalShortcut` to receive input sparingly.
- On Windows, possible type is `toolbar` .

Instance Events

Objects created with `new BrowserWindow` emit the following events:

Note: Some events are only available on specific operating systems and are labeled as such.

Event: 'page-title-updated'

Returns:

- `event` Event
- `title` String

Emitted when the document changed its title, calling `event.preventDefault()` will prevent the native window's title from changing.

Event: 'close'

Returns:

- `event` Event

Emitted when the window is going to be closed. It's emitted before the `beforeunload` and `unload` event of the DOM. Calling `event.preventDefault()` will cancel the close.

Usually you would want to use the `beforeunload` handler to decide whether the window should be closed, which will also be called when the window is reloaded. In Electron, returning any value other than `undefined` would cancel the close. For example:

```
window.onbeforeunload = (e) => {
  console.log('I do not want to be closed')

  // Unlike usual browsers that a message box will be prompted to users, returning
  // a non-void value will silently cancel the close.
  // It is recommended to use the dialog API to let the user confirm closing the
  // application.
  e.returnValue = false
}
```

Event: 'closed'

Emitted when the window is closed. After you have received this event you should remove the reference to the window and avoid using it any more.

Event: 'unresponsive'

Emitted when the web page becomes unresponsive.

Event: 'responsive'

Emitted when the unresponsive web page becomes responsive again.

Event: 'blur'

Emitted when the window loses focus.

Event: 'focus'

Emitted when the window gains focus.

Event: 'show'

Emitted when the window is shown.

Event: 'hide'

Emitted when the window is hidden.

Event: 'ready-to-show'

Emitted when the web page has been rendered and window can be displayed without a visual flash.

Event: 'maximize'

Emitted when window is maximized.

Event: 'unmaximize'

Emitted when the window exits from a maximized state.

Event: 'minimize'

Emitted when the window is minimized.

Event: 'restore'

Emitted when the window is restored from a minimized state.

Event: 'resize'

Emitted when the window is being resized.

Event: 'move'

Emitted when the window is being moved to a new position.

Note: On macOS this event is just an alias of `moved`.

Event: 'moved' *macOS*

Emitted once when the window is moved to a new position.

Event: 'enter-full-screen'

Emitted when the window enters a full-screen state.

Event: 'leave-full-screen'

Emitted when the window leaves a full-screen state.

Event: 'enter-html-full-screen'

Emitted when the window enters a full-screen state triggered by HTML API.

Event: 'leave-html-full-screen'

Emitted when the window leaves a full-screen state triggered by HTML API.

Event: 'app-command' *Windows*

Returns:

- `event` Event
- `command` String

Emitted when an App Command.aspx) is invoked. These are typically related to keyboard media keys or browser commands, as well as the "Back" button built into some mice on Windows.

Commands are lowercased, underscores are replaced with hyphens, and the `APPCOMMAND_` prefix is stripped off. e.g. `APPCOMMAND_BROWSER_BACKWARD` is emitted as `browser-backward`.

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()
win.on('app-command', (e, cmd) => {
  // Navigate the window back when the user hits their mouse back button
  if (cmd === 'browser-backward' && win.webContents.canGoBack()) {
    win.webContents.goBack()
  }
})
```

Event: 'scroll-touch-begin' *macOS*

Emitted when scroll wheel event phase has begun.

Event: 'scroll-touch-end' *macOS*

Emitted when scroll wheel event phase has ended.

Event: 'scroll-touch-edge' *macOS*

Emitted when scroll wheel event phase filed upon reaching the edge of element.

Event: 'swipe' *macOS*

Returns:

- `event` Event
- `direction` String

Emitted on 3-finger swipe. Possible directions are `up` , `right` , `down` , `left` .

Static Methods

The `BrowserWindow` class has the following static methods:

`BrowserWindow.getAllWindows()`

Returns `BrowserWindow[]` - An array of all opened browser windows.

`BrowserWindow.getFocusedWindow()`

Returns `BrowserWindow` - The window that is focused in this application, otherwise returns `null` .

`BrowserWindow.fromWebContents(webContents)`

- `webContents` WebContents

Returns `BrowserWindow` - The window that owns the given `webContents` .

`BrowserWindow.fromId(id)`

- `id` Integer

Returns `BrowserWindow` - The window with the given `id` .

`BrowserWindow.addDevToolsExtension(path)`

- `path` String

Adds DevTools extension located at `path` , and returns extension's name.

The extension will be remembered so you only need to call this API once, this API is not for programming use. If you try to add an extension that has already been loaded, this method will not return and instead log a warning to the console.

The method will also not return if the extension's manifest is missing or incomplete.

Note: This API cannot be called before the `ready` event of the `app` module is emitted.

`BrowserWindow.removeDevToolsExtension(name)`

- `name` String

Remove a DevTools extension by name.

Note: This API cannot be called before the `ready` event of the `app` module is emitted.

`BrowserWindow.getDevToolsExtensions()`

Returns `Object` - The keys are the extension names and each value is an Object containing `name` and `version` properties.

To check if a DevTools extension is installed you can run the following:

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

let installed = BrowserWindow.getDevToolsExtensions().hasOwnProperty('devtron')
console.log(installed)
```

Note: This API cannot be called before the `ready` event of the `app` module is emitted.

Instance Properties

Objects created with `new BrowserWindow` have the following properties:

```
const {BrowserWindow} = require('electron')
// In this example `win` is our instance
let win = new BrowserWindow({width: 800, height: 600})
win.loadURL('https://github.com')
```

`win.webContents`

A `WebContents` object this window owns. All web page related events and operations will be done via it.

See the `webContents` documentation for its methods and events.

`win.id`

A `Integer` representing the unique ID of the window.

Instance Methods

Objects created with `new BrowserWindow` have the following instance methods:

Note: Some methods are only available on specific operating systems and are labeled as such.

`win.destroy()`

Force closing the window, the `unload` and `beforeunload` event won't be emitted for the web page, and `close` event will also not be emitted for this window, but it guarantees the `closed` event will be emitted.

`win.close()`

Try to close the window. This has the same effect as a user manually clicking the close button of the window. The web page may cancel the close though. See the close event.

`win.focus()`

Focuses on the window.

`win.blur()`

Removes focus from the window.

`win.isFocused()`

Returns `Boolean` - Whether the window is focused.

`win.isDestroyed()`

Returns `Boolean` - Whether the window is destroyed.

`win.show()`

Shows and gives focus to the window.

`win.showInactive()`

Shows the window but doesn't focus on it.

`win.hide()`

Hides the window.

`win.isVisible()`

Returns `Boolean` - Whether the window is visible to the user.

`win.isModal()`

Returns `Boolean` - Whether current window is a modal window.

`win.maximize()`

Maximizes the window.

`win.unmaximize()`

Unmaximizes the window.

`win.isMaximized()`

Returns `Boolean` - Whether the window is maximized.

`win.minimize()`

Minimizes the window. On some platforms the minimized window will be shown in the Dock.

`win.restore()`

Restores the window from minimized state to its previous state.

`win.isMinimized()`

Returns `Boolean` - Whether the window is minimized.

`win.setFullScreen(flag)`

- `flag` Boolean

Sets whether the window should be in fullscreen mode.

`win.isFullScreen()`

Returns `Boolean` - Whether the window is in fullscreen mode.

`win.setAspectRatio(aspectRatio[, extraSize])` *macOS*

- `aspectRatio` Float - The aspect ratio to maintain for some portion of the content view.
- `extraSize` Object (optional) - The extra size not to be included while maintaining the aspect ratio.
  - `width` Integer
  - `height` Integer

This will make a window maintain an aspect ratio. The extra size allows a developer to have space, specified in pixels, not included within the aspect ratio calculations. This API already takes into account the difference between a window's size and its content size.

Consider a normal window with an HD video player and associated controls. Perhaps there are 15 pixels of controls on the left edge, 25 pixels of controls on the right edge and 50 pixels of controls below the player. In order to maintain a 16:9 aspect ratio (standard aspect ratio for HD @1920x1080) within the player itself we would call this function with arguments of 16/9 and [40, 50]. The second argument doesn't care where the extra width and height are within the content view--only that they exist. Just sum any extra width and height areas you have within the overall content view.

`win.previewFile(path[, displayName])` *macOS*

- `path` String - The absolute path to the file to preview with QuickLook. This is important as Quick Look uses the file name and file extension on the path to determine the content type of the file to open.
- `displayName` String (Optional) - The name of the file to display on the Quick Look modal view. This is purely visual and does not affect the content type of the file. Defaults to `path` .

Uses Quick Look to preview a file at a given path.

`win.closeFilePreview()` *macOS*

Closes the currently open Quick Look panel.

`win.setBounds(bounds[, animate])`

- `bounds` Rectangle
- `animate` Boolean (optional) *macOS*

Resizes and moves the window to the supplied bounds

`win.getBounds()`

Returns `Rectangle`

`win.setContentBounds(bounds[, animate])`

- `bounds` Rectangle

- `animate` Boolean (optional) *macOS*

Resizes and moves the window's client area (e.g. the web page) to the supplied bounds.

`win.getContentBounds()`

Returns `Rectangle`

`win.setSize(width, height[, animate])`

- `width` Integer
- `height` Integer
- `animate` Boolean (optional) *macOS*

Resizes the window to `width` and `height` .

`win.getSize()`

Returns `Integer[]` - Contains the window's width and height.

`win.setContentSize(width, height[, animate])`

- `width` Integer
- `height` Integer
- `animate` Boolean (optional) *macOS*

Resizes the window's client area (e.g. the web page) to `width` and `height` .

`win.getContentSize()`

Returns `Integer[]` - Contains the window's client area's width and height.

`win.setMinimumSize(width, height)`

- `width` Integer
- `height` Integer

Sets the minimum size of window to `width` and `height` .

`win.getMinimumSize()`

Returns `Integer[]` - Contains the window's minimum width and height.

`win.setMaximumSize(width, height)`

- `width` Integer
- `height` Integer

Sets the maximum size of window to `width` and `height` .

`win.getMaximumSize()`

Returns `Integer[]` - Contains the window's maximum width and height.

`win.setResizable(resizable)`

- `resizable` Boolean

Sets whether the window can be manually resized by user.

`win.isResizable()`

Returns `Boolean` - Whether the window can be manually resized by user.

`win.setMovable(movable)` *macOS Windows*

- `movable` Boolean

Sets whether the window can be moved by user. On Linux does nothing.

`win.isMovable()` *macOS Windows*

Returns `Boolean` - Whether the window can be moved by user.

On Linux always returns `true` .

`win.setMinimizable(minimizable)` *macOS Windows*

- `minimizable` Boolean

Sets whether the window can be manually minimized by user. On Linux does nothing.

`win.isMinimizable()` *macOS Windows*

Returns `Boolean` - Whether the window can be manually minimized by user

On Linux always returns `true` .

`win.setMaximizable(maximizable)` *macOS Windows*

- `maximizable` Boolean

Sets whether the window can be manually maximized by user. On Linux does nothing.

`win.isMaximizable()` *macOS Windows*

Returns `Boolean` - Whether the window can be manually maximized by user.

On Linux always returns `true` .

`win.setFullScreenable(fullscreenable)`

- `fullscreenable` Boolean

Sets whether the maximize/zoom window button toggles fullscreen mode or maximizes the window.

`win.isFullScreenable()`

Returns `Boolean` - Whether the maximize/zoom window button toggles fullscreen mode or maximizes the window.

`win.setClosable(closable)` *macOS Windows*

- `closable` Boolean

Sets whether the window can be manually closed by user. On Linux does nothing.

`win.isClosable()` *macOS Windows*

Returns `Boolean` - Whether the window can be manually closed by user.

On Linux always returns `true` .

`win.setAlwaysOnTop(flag[, level][, relativeLevel])`

- `flag` Boolean
- `level` String (optional) *macOS* - Values include `normal` , `floating` , `torn-off-menu` , `modal-panel` , `main-menu` , `status` , `pop-up-menu` , `screen-saver` , and ~~`dock`~~ (Deprecated). The default is `floating` . See the macOS docs for more details.
- `relativeLevel` Integer (optional) *macOS* - The number of layers higher to set this window relative to the given `level` . The default is `0` . Note that Apple discourages setting levels higher than 1 above `screen-saver` .

Sets whether the window should show always on top of other windows. After setting this, the window is still a normal window, not a toolbox window which can not be focused on.

`win.isAlwaysOnTop()`

Returns `Boolean` - Whether the window is always on top of other windows.

`win.center()`

Moves window to the center of the screen.

`win.setPosition(x, y[, animate])`

- `x` Integer
- `y` Integer
- `animate` Boolean (optional) *macOS*

Moves window to `x` and `y` .

`win.getPosition()`

Returns `Integer[]` - Contains the window's current position.

`win.setTitle(title)`

- `title` String

Changes the title of native window to `title` .

`win.getTitle()`

Returns `String` - The title of the native window.

Note: The title of web page can be different from the title of the native window.

`win.setSheetOffset(offsetY[, offsetX])` *macOS*

- `offsetY` Float
- `offsetX` Float (optional)

Changes the attachment point for sheets on macOS. By default, sheets are attached just below the window frame, but you may want to display them beneath a HTML-rendered toolbar. For example:

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()

let toolbarRect = document.getElementById('toolbar').getBoundingClientRect()
win.setSheetOffset(toolbarRect.height)
```

`win.flashFrame(flag)`

- `flag` Boolean

Starts or stops flashing the window to attract user's attention.

`win.setSkipTaskbar(skip)`

- `skip` Boolean

Makes the window not show in the taskbar.

`win.setKiosk(flag)`

- `flag` Boolean

Enters or leaves the kiosk mode.

`win.isKiosk()`

Returns `Boolean` - Whether the window is in kiosk mode.

`win.getNativeWindowHandle()`

Returns `Buffer` - The platform-specific handle of the window.

The native type of the handle is `HWND` on Windows, `NSView*` on macOS, and `Window` (`unsigned long`) on Linux.

`win.hookWindowMessage(message, callback)` *Windows*

- `message` Integer
- `callback` Function

Hooks a windows message. The `callback` is called when the message is received in the WndProc.

`win.isWindowMessageHooked(message)` *Windows*

- `message` Integer

Returns `Boolean` - `true` or `false` depending on whether the message is hooked.

`win.unhookWindowMessage(message)` *Windows*

- `message` Integer

Unhook the window message.

`win.unhookAllWindowMessages()` *Windows*

Unhooks all of the window messages.

`win.setRepresentedFilename(filename)` *macOS*

- `filename` String

Sets the pathname of the file the window represents, and the icon of the file will show in window's title bar.

`win.getRepresentedFilename()` *macOS*

Returns `String` - The pathname of the file the window represents.

`win.setDocumentEdited(edited)` *macOS*

- `edited` Boolean

Specifies whether the window’s document has been edited, and the icon in title bar will become gray when set to `true`.

`win.isDocumentEdited()` *macOS*

Returns `Boolean` - Whether the window's document has been edited.

`win.focusOnWebView()`

`win.blurWebView()`

`win.capturePage([rect, ]callback)`

- `rect` Rectangle (optional) - The bounds to capture
- `callback` Function
  - `image` NativeImage

Same as `webContents.capturePage([rect, ]callback)`.

`win.loadURL(url[, options])`

- `url` String
- `options` Object (optional)
  - `httpReferrer` String (optional) - A HTTP Referrer url.
  - `userAgent` String (optional) - A user agent originating the request.
  - `extraHeaders` String (optional) - Extra headers separated by "\n"
  - `postData` (UploadRawData | UploadFile | UploadFileSystem | UploadBlob)[] - (optional)
  - `baseURLForDataURL` String (optional) - Base url (with trailing path separator) for files to be loaded by the data url. This is needed only if the specified `url` is a data url and needs to load other files.

Same as `webContents.loadURL(url[, options])`.

The `url` can be a remote address (e.g. `http://`) or a path to a local HTML file using the `file://` protocol.

To ensure that file URLs are properly formatted, it is recommended to use Node's `url.format` method:

```
let url = require('url').format({
  protocol: 'file',
  slashes: true,
  pathname: require('path').join(__dirname, 'index.html')
})

win.loadURL(url)
```

You can load a URL using a `POST` request with URL-encoded data by doing the following:

```
win.loadURL('http://localhost:8000/post', {
  postData: [{
    type: 'rawData',
    bytes: Buffer.from('hello=world')
  }],
  extraHeaders: 'Content-Type: application/x-www-form-urlencoded'
})
```

`win.reload()`

Same as `webContents.reload` .

`win.setMenu(menu)` *Linux Windows*

- `menu` Menu

Sets the `menu` as the window's menu bar, setting it to `null` will remove the menu bar.

`win.setProgressBar(progress[, options])`

- `progress` Double
- `options` Object (optional)
  - `mode` String *Windows* - Mode for the progress bar. Can be `none` , `normal` , `indeterminate` , `error` , or `paused` .

Sets progress value in progress bar. Valid range is [0, 1.0].

Remove progress bar when progress < 0; Change to indeterminate mode when progress > 1.

On Linux platform, only supports Unity desktop environment, you need to specify the `*.desktop` file name to `desktopName` field in `package.json` . By default, it will assume `app.getName().desktop` .

On Windows, a mode can be passed. Accepted values are `none` , `normal` , `indeterminate` , `error` , and `paused` . If you call `setProgressBar` without a mode set (but with a value within the valid range), `normal` will be assumed.

`win.setOverlayIcon(overlay, description)` *Windows*

- `overlay` NativeImage - the icon to display on the bottom right corner of the taskbar icon. If this parameter is `null` , the overlay is cleared
- `description` String - a description that will be provided to Accessibility screen readers

Sets a 16 x 16 pixel overlay onto the current taskbar icon, usually used to convey some sort of application status or to passively notify the user.

`win.setHasShadow(hasShadow)` *macOS*

- `hasShadow` Boolean

Sets whether the window should have a shadow. On Windows and Linux does nothing.

`win.hasShadow()` *macOS*

Returns `Boolean` - Whether the window has a shadow.

On Windows and Linux always returns `true` .

`win.setThumbarButtons(buttons)` *Windows*

- `buttons` ThumbarButton[]

Returns `Boolean` - Whether the buttons were added successfully

Add a thumbnail toolbar with a specified set of buttons to the thumbnail image of a window in a taskbar button layout. Returns a `Boolean` object indicates whether the thumbnail has been added successfully.

The number of buttons in thumbnail toolbar should be no greater than 7 due to the limited room. Once you setup the thumbnail toolbar, the toolbar cannot be removed due to the platform's limitation. But you can call the API with an empty array to clean the buttons.

The `buttons` is an array of `Button` objects:

- `Button` Object
  - `icon` NativeImage - The icon showing in thumbnail toolbar.
  - `click` Function
  - `tooltip` String (optional) - The text of the button's tooltip.
  - `flags` String[] (optional) - Control specific states and behaviors of the button. By default, it is `['enabled']` .

The `flags` is an array that can include following `String` s:

- `enabled` - The button is active and available to the user.
- `disabled` - The button is disabled. It is present, but has a visual state indicating it will not respond to user action.
- `dismissonclick` - When the button is clicked, the thumbnail window closes immediately.
- `nobackground` - Do not draw a button border, use only the image.
- `hidden` - The button is not shown to the user.
- `noninteractive` - The button is enabled but not interactive; no pressed button state is drawn. This value is intended for instances where the button is used in a notification.

`win.setThumbnailClip(region)` *Windows*

- `region` Rectangle - Region of the window

Sets the region of the window to show as the thumbnail image displayed when hovering over the window in the taskbar. You can reset the thumbnail to be the entire window by specifying an empty region: `{x: 0, y: 0, width: 0, height: 0}` .

`win.setThumbnailToolTip(toolTip)` *Windows*

- `toolTip` String

Sets the toolTip that is displayed when hovering over the window thumbnail in the taskbar.

`win.setAppDetails(options)` *Windows*

- `options` Object
  - `appId` String (optional) - Window's App User Model ID.aspx). It has to be set, otherwise the other options will have no effect.
  - `appIconPath` String (optional) - Window's Relaunch Icon.aspx).
  - `appIconIndex` Integer (optional) - Index of the icon in `appIconPath` . Ignored when `appIconPath` is not set. Default is `0` .
  - `relaunchCommand` String (optional) - Window's Relaunch Command.aspx).
  - `relaunchDisplayName` String (optional) - Window's Relaunch Display Name.aspx).

Sets the properties for the window's taskbar button.

Note: `relaunchCommand` and `relaunchDisplayName` must always be set together. If one of those properties is not set, then neither will be used.

`win.showDefinitionForSelection()` *macOS*

Same as `webContents.showDefinitionForSelection()` .

`win.setIcon(icon)` *Windows Linux*

- `icon` NativeImage

Changes window icon.

`win.setAutoHideMenuBar(hide)`

- `hide` Boolean

Sets whether the window menu bar should hide itself automatically. Once set the menu bar will only show when users press the single `Alt` key.

If the menu bar is already visible, calling `setAutoHideMenuBar(true)` won't hide it immediately.

`win.isMenuBarAutoHide()`

Returns `Boolean` - Whether menu bar automatically hides itself.

`win.setMenuBarVisibility(visible)` *Windows Linux*

- `visible` Boolean

Sets whether the menu bar should be visible. If the menu bar is auto-hide, users can still bring up the menu bar by pressing the single `Alt` key.

`win.isMenuBarVisible()`

Returns `Boolean` - Whether the menu bar is visible.

`win.setVisibleOnAllWorkspaces(visible)`

- `visible` Boolean

Sets whether the window should be visible on all workspaces.

Note: This API does nothing on Windows.

`win.isVisibleOnAllWorkspaces()`

Returns `Boolean` - Whether the window is visible on all workspaces.

Note: This API always returns false on Windows.

`win.setIgnoreMouseEvents(ignore)`

- `ignore` Boolean

Makes the window ignore all mouse events.

All mouse events happened in this window will be passed to the window below this window, but if this window has focus, it will still receive keyboard events.

`win.setContentProtection(enable)` *macOS Windows*

- `enable` Boolean

Prevents the window contents from being captured by other apps.

On macOS it sets the NSWindow's sharingType to NSWindowSharingNone. On Windows it calls SetWindowDisplayAffinity with `WDA_MONITOR`.

`win.setFocusable(focusable)` *Windows*

- `focusable` Boolean

Changes whether the window can be focused.

`win.setParentWindow(parent)` *Linux macOS*

- `parent` BrowserWindow

Sets `parent` as current window's parent window, passing `null` will turn current window into a top-level window.

`win.getParentWindow()`

Returns `BrowserWindow` - The parent window.

`win.getChildWindows()`

Returns `BrowserWindow[]` - All child windows.

`win.setAutoHideCursor(autoHide)` *macOS*

- `autoHide` Boolean

Controls whether to hide cursor when typing.

`win.setVibrancy(type)` *macOS*

- `type` String - Can be `appearance-based`, `light`, `dark`, `titlebar`, `selection`, `menu`, `popover`, `sidebar`, `medium-light` or `ultra-dark`. See the macOS documentation for more details.

Adds a vibrancy effect to the browser window. Passing `null` or an empty string will remove the vibrancy effect on the window.

`win.setTouchBar(touchBar)` *macOS*

- `touchBar` TouchBar

Sets the touchBar layout for the current window. Specifying `null` or `undefined` clears the touch bar. This method only has an effect if the machine has a touch bar and is running on macOS 10.12.1+.

Note: The TouchBar API is currently experimental and may change or be removed in future Electron releases.

contentTracing

> Collect tracing data from Chromium's content module for finding performance bottlenecks and slow operations.

Process: Main

This module does not include a web interface so you need to open `chrome://tracing/` in a Chrome browser and load the generated file to view the result.

Note: You should not use this module until the `ready` event of the app module is emitted.

```
const {app, contentTracing} = require('electron')

app.on('ready', () => {
  const options = {
    categoryFilter: '*',
    traceOptions: 'record-until-full,enable-sampling'
  }

  contentTracing.startRecording(options, () => {
    console.log('Tracing started')

    setTimeout(() => {
      contentTracing.stopRecording('', (path) => {
        console.log('Tracing data recorded to ' + path)
      })
    }, 5000)
  })
})
```

Methods

The `contentTracing` module has the following methods:

`contentTracing.getCategories(callback)`

- `callback` Function
  - `categories` String[]

Get a set of category groups. The category groups can change as new code paths are reached.

Once all child processes have acknowledged the `getCategories` request the `callback` is invoked with an array of category groups.

`contentTracing.startRecording(options, callback)`

- `options` Object
  - `categoryFilter` String
  - `traceOptions` String
- `callback` Function

Start recording on all processes.

Recording begins immediately locally and asynchronously on child processes as soon as they receive the EnableRecording request. The `callback` will be called once all child processes have acknowledged the `startRecording` request.

`categoryFilter` is a filter to control what category groups should be traced. A filter can have an optional `-` prefix to exclude category groups that contain a matching category. Having both included and excluded category patterns in the same list is not supported.

Examples:

- `test_MyTest*` ,
- `test_MyTest*,test_OtherStuff` ,
- `"-excluded_category1,-excluded_category2`

`traceOptions` controls what kind of tracing is enabled, it is a comma-delimited list. Possible options are:

- `record-until-full`
- `record-continuously`
- `trace-to-console`
- `enable-sampling`
- `enable-systrace`

The first 3 options are trace recording modes and hence mutually exclusive. If more than one trace recording modes appear in the `traceOptions` string, the last one takes precedence. If none of the trace recording modes are specified, recording mode is `record-until-full` .

The trace option will first be reset to the default option (`record_mode` set to `record-until-full` , `enable_sampling` and `enable_systrace` set to `false`) before options parsed from `traceOptions` are applied on it.

`contentTracing.stopRecording(resultFilePath, callback)`

- `resultFilePath` String
- `callback` Function
  - `resultFilePath` String

Stop recording on all processes.

Child processes typically cache trace data and only rarely flush and send trace data back to the main process. This helps to minimize the runtime overhead of tracing since sending trace data over IPC can be an expensive operation. So, to end tracing, we must asynchronously ask all child processes to flush any pending trace data.

Once all child processes have acknowledged the `stopRecording` request, `callback` will be called with a file that contains the traced data.

Trace data will be written into `resultFilePath` if it is not empty or into a temporary file. The actual file path will be passed to `callback` if it's not `null` .

`contentTracing.startMonitoring(options, callback)`

- `options` Object
  - `categoryFilter` String
  - `traceOptions` String
- `callback` Function

Start monitoring on all processes.

Monitoring begins immediately locally and asynchronously on child processes as soon as they receive the `startMonitoring` request.

Once all child processes have acknowledged the `startMonitoring` request the `callback` will be called.

`contentTracing.stopMonitoring(callback)`

- `callback` Function

Stop monitoring on all processes.

Once all child processes have acknowledged the `stopMonitoring` request the `callback` is called.

`contentTracing.captureMonitoringSnapshot(resultFilePath, callback)`

- `resultFilePath` String
- `callback` Function
  - `resultFilePath` String

Get the current monitoring traced data.

Child processes typically cache trace data and only rarely flush and send trace data back to the main process. This is because it may be an expensive operation to send the trace data over IPC and we would like to avoid unneeded runtime overhead from tracing. So, to end tracing, we must asynchronously ask all child processes to flush any pending trace data.

Once all child processes have acknowledged the `captureMonitoringSnapshot` request the `callback` will be called with a file that contains the traced data.

`contentTracing.getTraceBufferUsage(callback)`

- `callback` Function
  - `value` Number
  - `percentage` Number

Get the maximum usage across processes of trace buffer as a percentage of the full state. When the TraceBufferUsage value is determined the `callback` is called.

dialog

> Display native system dialogs for opening and saving files, alerting, etc.

Process: Main

An example of showing a dialog to select multiple files and directories:

```
const {dialog} = require('electron')
console.log(dialog.showOpenDialog({properties: ['openFile', 'openDirectory', 'multiSelections']}))
```

The Dialog is opened from Electron's main thread. If you want to use the dialog object from a renderer process, remember to access it using the remote:

```
const {dialog} = require('electron').remote
console.log(dialog)
```

Methods

The `dialog` module has the following methods:

`dialog.showOpenDialog([browserWindow, ]options[, callback])`

- `browserWindow` BrowserWindow (optional)
- `options` Object
  - `title` String (optional)
  - `defaultPath` String (optional)
  - `buttonLabel` String (optional) - Custom label for the confirmation button, when left empty the default label will be used.
  - `filters` FileFilter[] (optional)
  - `properties` String[] (optional) - Contains which features the dialog should use. The following values are supported:
    - `openFile` - Allow files to be selected.
    - `openDirectory` - Allow directories to be selected.
    - `multiSelections` - Allow multiple paths to be selected.
    - `showHiddenFiles` - Show hidden files in dialog.
    - `createDirectory` - Allow creating new directories from dialog. *macOS*
    - `promptToCreate` - Prompt for creation if the file path entered in the dialog does not exist. This does not actually create the file at the path but allows non-existent paths to be returned that should be created by the application. *Windows*
    - `noResolveAliases` - Disable the automatic alias (symlink) path resolution. Selected aliases will now return the alias path instead of their target path. *macOS*
  - `normalizeAccessKeys` Boolean (optional) - Normalize the keyboard access keys across platforms. Default is `false`. Enabling this assumes `&` is used in the button labels for the placement of the keyboard shortcut access key and labels will be converted so they work correctly on each platform, `&` characters are removed on macOS, converted to `_` on Linux, and left untouched on Windows. For example, a button label of `Vie&w` will be converted to `Vie_w` on Linux and `View` on macOS and can be selected via `Alt-W` on Windows and Linux.
    - `message` String (optional) *macOS* - Message to display above input boxes.
- `callback` Function (optional)
  - `filePaths` String[] - An array of file paths chosen by the user

Returns `String[]`, an array of file paths chosen by the user, if the callback is provided it returns `undefined`.

The `browserWindow` argument allows the dialog to attach itself to a parent window, making it modal.

The `filters` specifies an array of file types that can be displayed or selected when you want to limit the user to a specific type. For example:

```
{
  filters: [
    {name: 'Images', extensions: ['jpg', 'png', 'gif']},
    {name: 'Movies', extensions: ['mkv', 'avi', 'mp4']},
    {name: 'Custom File Type', extensions: ['as']},
    {name: 'All Files', extensions: ['*']}
  ]
}
```

The `extensions` array should contain extensions without wildcards or dots (e.g. `'png'` is good but `'.png'` and `'*.png'` are bad). To show all files, use the `'*'` wildcard (no other wildcard is supported).

If a `callback` is passed, the API call will be asynchronous and the result will be passed via `callback(filenames)`

Note: On Windows and Linux an open dialog can not be both a file selector and a directory selector, so if you set `properties` to `['openFile', 'openDirectory']` on these platforms, a directory selector will be shown.

`dialog.showSaveDialog([browserWindow, ]options[, callback])`

- `browserWindow` BrowserWindow (optional)
- `options` Object
  - `title` String (optional)
  - `defaultPath` String (optional)
  - `buttonLabel` String (optional) - Custom label for the confirmation button, when left empty the default label will be used.
  - `filters` FileFilter[] (optional)
  - `message` String (optional) *macOS* - Message to display above text fields.
  - `nameFieldLabel` String (optional) *macOS* - Custom label for the text displayed in front of the filename text field.
  - `showsTagField` Boolean (optional) *macOS* - Show the tags input box, defaults to `true` .
- `callback` Function (optional)
  - `filename` String

Returns `String` , the path of the file chosen by the user, if a callback is provided it returns `undefined` .

The `browserWindow` argument allows the dialog to attach itself to a parent window, making it modal.

The `filters` specifies an array of file types that can be displayed, see `dialog.showOpenDialog` for an example.

If a `callback` is passed, the API call will be asynchronous and the result will be passed via `callback(filename)`

`dialog.showMessageBox([browserWindow, ]options[, callback])`

- `browserWindow` BrowserWindow (optional)
- `options` Object
  - `type` String (optional) - Can be `"none"` , `"info"` , `"error"` , `"question"` or `"warning"` . On Windows, "question" displays the same icon as "info", unless you set an icon using the "icon" option.
  - `buttons` String[] (optional) - Array of texts for buttons. On Windows, an empty array will result in one button labeled "OK".
  - `defaultId` Integer (optional) - Index of the button in the buttons array which will be selected by default when the message box opens.
  - `title` String (optional) - Title of the message box, some platforms will not show it.
  - `message` String - Content of the message box.
  - `detail` String (optional) - Extra information of the message.

  - `checkboxLabel` String (optional) - If provided, the message box will include a checkbox with the given label. The checkbox state can be inspected only when using `callback`.
  - `checkboxChecked` Boolean (optional) - Initial checked state of the checkbox. `false` by default.
  - `icon` NativeImage (optional)
  - `cancelId` Integer (optional) - The index of the button to be used to cancel the dialog, via the `Esc` key. By default this is assigned to the first button with "cancel" or "no" as the label. If no such labeled buttons exist and this option is not set, `0` will be used as the return value or callback response. This option is ignored on Windows.
  - `noLink` Boolean (optional) - On Windows Electron will try to figure out which one of the `buttons` are common buttons (like "Cancel" or "Yes"), and show the others as command links in the dialog. This can make the dialog appear in the style of modern Windows apps. If you don't like this behavior, you can set `noLink` to `true`.
- `callback` Function (optional)
  - `response` Number - The index of the button that was clicked
  - `checkboxChecked` Boolean - The checked state of the checkbox if `checkboxLabel` was set. Otherwise `false`.

Returns `Integer`, the index of the clicked button, if a callback is provided it returns undefined.

Shows a message box, it will block the process until the message box is closed. It returns the index of the clicked button.

The `browserWindow` argument allows the dialog to attach itself to a parent window, making it modal.

If a `callback` is passed, the API call will be asynchronous and the result will be passed via `callback(response)`.

`dialog.showErrorBox(title, content)`

- `title` String - The title to display in the error box
- `content` String - The text content to display in the error box

Displays a modal dialog that shows an error message.

This API can be called safely before the `ready` event the `app` module emits, it is usually used to report errors in early stage of startup. If called before the app `ready` event on Linux, the message will be emitted to stderr, and no GUI dialog will appear.

Sheets

On macOS, dialogs are presented as sheets attached to a window if you provide a `BrowserWindow` reference in the `browserWindow` parameter, or modals if no window is provided.

You can call `BrowserWindow.getCurrentWindow().setSheetOffset(offset)` to change the offset from the window frame where sheets are attached.

globalShortcut

> Detect keyboard events when the application does not have keyboard focus.

Process: Main

The `globalShortcut` module can register/unregister a global keyboard shortcut with the operating system so that you can customize the operations for various shortcuts.

Note: The shortcut is global; it will work even if the app does not have the keyboard focus. You should not use this module until the `ready` event of the app module is emitted.

```
const {app, globalShortcut} = require('electron')

app.on('ready', () => {
  // Register a 'CommandOrControl+X' shortcut listener.
  const ret = globalShortcut.register('CommandOrControl+X', () => {
    console.log('CommandOrControl+X is pressed')
  })

  if (!ret) {
    console.log('registration failed')
  }

  // Check whether a shortcut is registered.
  console.log(globalShortcut.isRegistered('CommandOrControl+X'))
})

app.on('will-quit', () => {
  // Unregister a shortcut.
  globalShortcut.unregister('CommandOrControl+X')

  // Unregister all shortcuts.
  globalShortcut.unregisterAll()
})
```

Methods

The `globalShortcut` module has the following methods:

`globalShortcut.register(accelerator, callback)`

- `accelerator` Accelerator
- `callback` Function

Registers a global shortcut of `accelerator` . The `callback` is called when the registered shortcut is pressed by the user.

When the accelerator is already taken by other applications, this call will silently fail. This behavior is intended by operating systems, since they don't want applications to fight for global shortcuts.

`globalShortcut.isRegistered(accelerator)`

- `accelerator` Accelerator

Returns `Boolean` - Whether this application has registered `accelerator` .

When the accelerator is already taken by other applications, this call will still return `false` . This behavior is intended by operating systems, since they don't want applications to fight for global shortcuts.

`globalShortcut.unregister(accelerator)`

- `accelerator` Accelerator

Unregisters the global shortcut of `accelerator`.

`globalShortcut.unregisterAll()`

Unregisters all of the global shortcuts.

ipcMain

> Communicate asynchronously from the main process to renderer processes.

Process: Main

The `ipcMain` module is an instance of the EventEmitter class. When used in the main process, it handles asynchronous and synchronous messages sent from a renderer process (web page). Messages sent from a renderer will be emitted to this module.

Sending Messages

It is also possible to send messages from the main process to the renderer process, see webContents.send for more information.

- When sending a message, the event name is the `channel`.
- To reply a synchronous message, you need to set `event.returnValue`.
- To send an asynchronous back to the sender, you can use `event.sender.send(...)`.

An example of sending and handling messages between the render and main processes:

```
// In main process.
const {ipcMain} = require('electron')
ipcMain.on('asynchronous-message', (event, arg) => {
  console.log(arg)  // prints "ping"
  event.sender.send('asynchronous-reply', 'pong')
})

ipcMain.on('synchronous-message', (event, arg) => {
  console.log(arg)  // prints "ping"
  event.returnValue = 'pong'
})
```

```
// In renderer process (web page).
const {ipcRenderer} = require('electron')
console.log(ipcRenderer.sendSync('synchronous-message', 'ping')) // prints "pong"

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

Methods

The `ipcMain` module has the following method to listen for events:

`ipcMain.on(channel, listener)`

- `channel` String
- `listener` Function

Listens to `channel`, when a new message arrives `listener` would be called with `listener(event, args...)`.

`ipcMain.once(channel, listener)`

- `channel` String

- `listener` Function

Adds a one time `listener` function for the event. This `listener` is invoked only the next time a message is sent to `channel` , after which it is removed.

`ipcMain.removeListener(channel, listener)`

- `channel` String
- `listener` Function

Removes the specified `listener` from the listener array for the specified `channel` .

`ipcMain.removeAllListeners([channel])`

- `channel` String (optional)

Removes all listeners, or those of the specified `channel` .

Event object

The `event` object passed to the `callback` has the following methods:

`event.returnValue`

Set this to the value to be returned in a synchronous message.

`event.sender`

Returns the `webContents` that sent the message, you can call `event.sender.send` to reply to the asynchronous message, see webContents.send for more information.

Class: Menu

> Create native application menus and context menus.

Process: Main

`new Menu()`

Creates a new menu.

Static Methods

The `menu` class has the following static methods:

`Menu.setApplicationMenu(menu)`

- `menu` Menu

Sets `menu` as the application menu on macOS. On Windows and Linux, the `menu` will be set as each window's top menu.

Note: This API has to be called after the `ready` event of `app` module.

`Menu.getApplicationMenu()`

Returns `Menu` - The application menu, if set, or `null` , if not set.

`Menu.sendActionToFirstResponder(action)` *macOS*

- `action` String

Sends the `action` to the first responder of application. This is used for emulating default Cocoa menu behaviors, usually you would just use the `role` property of `MenuItem` .

See the macOS Cocoa Event Handling Guide for more information on macOS' native actions.

`Menu.buildFromTemplate(template)`

- `template` MenuItemConstructorOptions[]

Returns `Menu`

Generally, the `template` is just an array of `options` for constructing a MenuItem. The usage can be referenced above.

You can also attach other fields to the element of the `template` and they will become properties of the constructed menu items.

Instance Methods

The `menu` object has the following instance methods:

`menu.popup([browserWindow, options])`

- `browserWindow` BrowserWindow (optional) - Default is the focused window.
- `options` Object (optional)
  - `x` Number (optional) - Default is the current mouse cursor position.
  - `y` Number (**required** if `x` is used) - Default is the current mouse cursor position.
  - `async` Boolean (optional) - Set to `true` to have this method return immediately called, `false` to return after the

menu has been selected or closed. Defaults to `false` .
  - `positioningItem` Number (optional) *macOS* - The index of the menu item to be positioned under the mouse cursor at the specified coordinates. Default is -1.

Pops up this menu as a context menu in the `browserWindow` .

`menu.closePopup([browserWindow])`

- `browserWindow` BrowserWindow (optional) - Default is the focused window.

Closes the context menu in the `browserWindow` .

`menu.append(menuItem)`

- `menuItem` MenuItem

Appends the `menuItem` to the menu.

`menu.insert(pos, menuItem)`

- `pos` Integer
- `menuItem` MenuItem

Inserts the `menuItem` to the `pos` position of the menu.

Instance Properties

`menu` objects also have the following properties:

`menu.items`

A MenuItem[] array containing the menu's items.

Each `Menu` consists of multiple `MenuItem` s and each `MenuItem` can have a submenu.

Examples

The `Menu` class is only available in the main process, but you can also use it in the render process via the `remote` module.

Main process

An example of creating the application menu in the main process with the simple template API:

```
const {app, Menu} = require('electron')

const template = [
  {
    label: 'Edit',
    submenu: [
      {
        role: 'undo'
      },
      {
        role: 'redo'
      },
      {
        type: 'separator'
      },
      {
        role: 'cut'
      },
      {
        role: 'copy'
      },
      {
        role: 'paste'
      },
      {
        role: 'pasteandmatchstyle'
      },
      {
        role: 'delete'
      },
      {
        role: 'selectall'
      }
    ]
  },
  {
    label: 'View',
    submenu: [
      {
        role: 'reload'
      },
      {
        role: 'forcereload'
      },
      {
        role: 'toggledevtools'
      },
      {
        type: 'separator'
      },
      {
        role: 'resetzoom'
      },
      {
        role: 'zoomin'
      },
      {
        role: 'zoomout'
      },
      {
        type: 'separator'
      },
      {
        role: 'togglefullscreen'
      }
    ]
  },
  {
    role: 'window',
    submenu: [
      {
        role: 'minimize'
      },
      {
        role: 'close'
      }
    ]
  },
  {
    role: 'help',
    submenu: [
      {
        label: 'Learn More',
        click () { require('electron').shell.openExternal('https://electron.atom.io') }
      }
    ]
  }
]
```

```
if (process.platform === 'darwin') {
  template.unshift({
    label: app.getName(),
    submenu: [
      {
        role: 'about'
      },
      {
        type: 'separator'
      },
      {
        role: 'services',
        submenu: []
      },
      {
        type: 'separator'
      },
      {
        role: 'hide'
      },
      {
        role: 'hideothers'
      },
      {
        role: 'unhide'
      },
      {
        type: 'separator'
      },
      {
        role: 'quit'
      }
    ]
  })
  // Edit menu.
  template[1].submenu.push(
    {
      type: 'separator'
    },
    {
      label: 'Speech',
      submenu: [
        {
          role: 'startspeaking'
        },
        {
          role: 'stopspeaking'
        }
      ]
    }
  )
  // Window menu.
  template[3].submenu = [
    {
      label: 'Close',
      accelerator: 'CmdOrCtrl+W',
      role: 'close'
    },
    {
      label: 'Minimize',
      accelerator: 'CmdOrCtrl+M',
      role: 'minimize'
    },
    {
      label: 'Zoom',
      role: 'zoom'
    },
    {
      type: 'separator'
    },
    {
      label: 'Bring All to Front',
      role: 'front'
    }
  ]
}

const menu = Menu.buildFromTemplate(template)
Menu.setApplicationMenu(menu)
```

Render process

Below is an example of creating a menu dynamically in a web page (render process) by using the `remote` module, and showing it when the user right clicks the page:

```
<!-- index.html -->
<script>
const {remote} = require('electron')
const {Menu, MenuItem} = remote

const menu = new Menu()
menu.append(new MenuItem({label: 'MenuItem1', click() { console.log('item 1 clicked') }}))
menu.append(new MenuItem({type: 'separator'}))
menu.append(new MenuItem({label: 'MenuItem2', type: 'checkbox', checked: true}))

window.addEventListener('contextmenu', (e) => {
  e.preventDefault()
  menu.popup(remote.getCurrentWindow())
}, false)
</script>
```

Notes on macOS Application Menu

macOS has a completely different style of application menu from Windows and Linux. Here are some notes on making your app's menu more native-like.

Standard Menus

On macOS there are many system-defined standard menus, like the `Services` and `Windows` menus. To make your menu a standard menu, you should set your menu's `role` to one of following and Electron will recognize them and make them become standard menus:

- `window`
- `help`
- `services`

Standard Menu Item Actions

macOS has provided standard actions for some menu items, like `About xxx` , `Hide xxx` , and `Hide Others` . To set the action of a menu item to a standard action, you should set the `role` attribute of the menu item.

Main Menu's Name

On macOS the label of the application menu's first item is always your app's name, no matter what label you set. To change it, modify your app bundle's `Info.plist` file. See About Information Property List Files for more information.

Setting Menu for Specific Browser Window (*Linux Windows*)

The `setMenu` method of browser windows can set the menu of certain browser windows.

Menu Item Position

You can make use of `position` and `id` to control how the item will be placed when building a menu with `Menu.buildFromTemplate`.

The `position` attribute of `MenuItem` has the form `[placement]=[id]`, where `placement` is one of `before`, `after`, or `endof` and `id` is the unique ID of an existing item in the menu:

- `before` - Inserts this item before the id referenced item. If the referenced item doesn't exist the item will be inserted at the end of the menu.
- `after` - Inserts this item after id referenced item. If the referenced item doesn't exist the item will be inserted at the end of the menu.
- `endof` - Inserts this item at the end of the logical group containing the id referenced item (groups are created by separator items). If the referenced item doesn't exist, a new separator group is created with the given id and this item is inserted after that separator.

When an item is positioned, all un-positioned items are inserted after it until a new item is positioned. So if you want to position a group of menu items in the same location you only need to specify a position for the first item.

Examples

Template:

```
[
  {label: '4', id: '4'},
  {label: '5', id: '5'},
  {label: '1', id: '1', position: 'before=4'},
  {label: '2', id: '2'},
  {label: '3', id: '3'}
]
```

Menu:

```
- 1
- 2
- 3
- 4
- 5
```

Template:

```
[
  {label: 'a', position: 'endof=letters'},
  {label: '1', position: 'endof=numbers'},
  {label: 'b', position: 'endof=letters'},
  {label: '2', position: 'endof=numbers'},
  {label: 'c', position: 'endof=letters'},
  {label: '3', position: 'endof=numbers'}
]
```

Menu:

```
- ---
- a
- b
- c
- ---
- 1
- 2
- 3
```

Class: MenuItem

> Add items to native application menus and context menus.

Process: Main

See `Menu` for examples.

`new MenuItem(options)`

- `options` Object
  - `click` Function (optional) - Will be called with `click(menuItem, browserWindow, event)` when the menu item is clicked.
    - `menuItem` MenuItem
    - `browserWindow` BrowserWindow
    - `event` Event
  - `role` String (optional) - Define the action of the menu item, when specified the `click` property will be ignored.
  - `type` String (optional) - Can be `normal` , `separator` , `submenu` , `checkbox` or `radio` .
  - `label` String - (optional)
  - `sublabel` String - (optional)
  - `accelerator` Accelerator (optional)
  - `icon` (NativeImage | String) (optional)
  - `enabled` Boolean (optional) - If false, the menu item will be greyed out and unclickable.
  - `visible` Boolean (optional) - If false, the menu item will be entirely hidden.
  - `checked` Boolean (optional) - Should only be specified for `checkbox` or `radio` type menu items.
  - `submenu` (MenuItemConstructorOptions[] | Menu) (optional) - Should be specified for `submenu` type menu items. If `submenu` is specified, the `type: 'submenu'` can be omitted. If the value is not a `Menu` then it will be automatically converted to one using `Menu.buildFromTemplate` .
  - `id` String (optional) - Unique within a single menu. If defined then it can be used as a reference to this item by the position attribute.
  - `position` String (optional) - This field allows fine-grained definition of the specific location within a given menu.

It is best to specify `role` for any menu item that matches a standard role, rather than trying to manually implement the behavior in a `click` function. The built-in `role` behavior will give the best native experience.

The `label` and `accelerator` are optional when using a `role` and will default to appropriate values for each platform.

The `role` property can have following values:

- `undo`
- `redo`
- `cut`
- `copy`
- `paste`
- `pasteandmatchstyle`
- `selectall`
- `delete`
- `minimize` - Minimize current window
- `close` - Close current window
- `quit` - Quit the application
- `reload` - Reload the current window
- `forcereload` - Reload the current window ignoring the cache.
- `toggledevtools` - Toggle developer tools in the current window
- `togglefullscreen` - Toggle full screen mode on the current window

- `resetzoom` - Reset the focused page's zoom level to the original size
- `zoomin` - Zoom in the focused page by 10%
- `zoomout` - Zoom out the focused page by 10%

On macOS `role` can also have following additional values:

- `about` - Map to the `orderFrontStandardAboutPanel` action
- `hide` - Map to the `hide` action
- `hideothers` - Map to the `hideOtherApplications` action
- `unhide` - Map to the `unhideAllApplications` action
- `startspeaking` - Map to the `startSpeaking` action
- `stopspeaking` - Map to the `stopSpeaking` action
- `front` - Map to the `arrangeInFront` action
- `zoom` - Map to the `performZoom` action
- `window` - The submenu is a "Window" menu
- `help` - The submenu is a "Help" menu
- `services` - The submenu is a "Services" menu

When specifying `role` on macOS, `label` and `accelerator` are the only options that will affect the MenuItem. All other options will be ignored.

Instance Properties

The following properties are available on instances of `MenuItem`:

`menuItem.enabled`

A Boolean indicating whether the item is enabled, this property can be dynamically changed.

`menuItem.visible`

A Boolean indicating whether the item is visible, this property can be dynamically changed.

`menuItem.checked`

A Boolean indicating whether the item is checked, this property can be dynamically changed.

A `checkbox` menu item will toggle the `checked` property on and off when selected.

A `radio` menu item will turn on its `checked` property when clicked, and will turn off that property for all adjacent items in the same menu.

You can add a `click` function for additional behavior.

`menuItem.label`

A String representing the menu items visible label

`menuItem.click`

A Function that is fired when the MenuItem recieves a click event

powerMonitor

> Monitor power state changes.

Process: Main

You cannot require or use this module until the `ready` event of the `app` module is emitted.

For example:

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

app.on('ready', () => {
  electron.powerMonitor.on('suspend', () => {
    console.log('The system is going to sleep')
  })
})
```

Events

The `powerMonitor` module emits the following events:

Event: 'suspend'

Emitted when the system is suspending.

Event: 'resume'

Emitted when system is resuming.

Event: 'on-ac' *Windows*

Emitted when the system changes to AC power.

Event: 'on-battery' *Windows*

Emitted when system changes to battery power.

powerSaveBlocker

> Block the system from entering low-power (sleep) mode.

Process: Main

For example:

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

const id = powerSaveBlocker.start('prevent-display-sleep')
console.log(powerSaveBlocker.isStarted(id))

powerSaveBlocker.stop(id)
```

Methods

The `powerSaveBlocker` module has the following methods:

`powerSaveBlocker.start(type)`

- `type` String - Power save blocker type.
    - `prevent-app-suspension` - Prevent the application from being suspended. Keeps system active but allows screen to be turned off. Example use cases: downloading a file or playing audio.
    - `prevent-display-sleep` - Prevent the display from going to sleep. Keeps system and screen active. Example use case: playing video.

Returns `Integer` - The blocker ID that is assigned to this power blocker

Starts preventing the system from entering lower-power mode. Returns an integer identifying the power save blocker.

Note: `prevent-display-sleep` has higher precedence over `prevent-app-suspension`. Only the highest precedence type takes effect. In other words, `prevent-display-sleep` always takes precedence over `prevent-app-suspension`.

For example, an API calling A requests for `prevent-app-suspension`, and another calling B requests for `prevent-display-sleep`. `prevent-display-sleep` will be used until B stops its request. After that, `prevent-app-suspension` is used.

`powerSaveBlocker.stop(id)`

- `id` Integer - The power save blocker id returned by `powerSaveBlocker.start`.

Stops the specified power save blocker.

`powerSaveBlocker.isStarted(id)`

- `id` Integer - The power save blocker id returned by `powerSaveBlocker.start`.

Returns `Boolean` - Whether the corresponding `powerSaveBlocker` has started.

protocol

> Register a custom protocol and intercept existing protocol requests.

Process: Main

An example of implementing a protocol that has the same effect as the `file://` protocol:

```
const {app, protocol} = require('electron')
const path = require('path')

app.on('ready', () => {
  protocol.registerFileProtocol('atom', (request, callback) => {
    const url = request.url.substr(7)
    callback({path: path.normalize(`${__dirname}/${url}`)})
  }, (error) => {
    if (error) console.error('Failed to register protocol')
  })
})
```

Note: All methods unless specified can only be used after the `ready` event of the `app` module gets emitted.

Methods

The `protocol` module has the following methods:

`protocol.registerStandardSchemes(schemes[, options])`

- `schemes` String[] - Custom schemes to be registered as standard schemes.
- `options` Object (optional)
  - `secure` Boolean (optional) - `true` to register the scheme as secure. Default `false` .

A standard scheme adheres to what RFC 3986 calls generic URI syntax. For example `http` and `https` are standard schemes, while `file` is not.

Registering a scheme as standard, will allow relative and absolute resources to be resolved correctly when served. Otherwise the scheme will behave like the `file` protocol, but without the ability to resolve relative URLs.

For example when you load following page with custom protocol without registering it as standard scheme, the image will not be loaded because non-standard schemes can not recognize relative URLs:

```
<body>
  <img src='test.png'>
</body>
```

Registering a scheme as standard will allow access to files through the FileSystem API. Otherwise the renderer will throw a security error for the scheme.

By default web storage apis (localStorage, sessionStorage, webSQL, indexedDB, cookies) are disabled for non standard schemes. So in general if you want to register a custom protocol to replace the `http` protocol, you have to register it as a standard scheme:

```
const {app, protocol} = require('electron')

protocol.registerStandardSchemes(['atom'])
app.on('ready', () => {
  protocol.registerHttpProtocol('atom', '...')
})
```

Note: This method can only be used before the `ready` event of the `app` module gets emitted.

`protocol.registerServiceWorkerSchemes(schemes)`

- `schemes` String[] - Custom schemes to be registered to handle service workers.

`protocol.registerFileProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
  - `request` Object
    - `url` String
    - `referrer` String
    - `method` String
    - `uploadData` UploadData[]
  - `callback` Function
    - `filePath` String (optional)
- `completion` Function (optional)
  - `error` Error

Registers a protocol of `scheme` that will send the file as a response. The `handler` will be called with `handler(request, callback)` when a `request` is going to be created with `scheme`. `completion` will be called with `completion(null)` when `scheme` is successfully registered or `completion(error)` when failed.

To handle the `request`, the `callback` should be called with either the file's path or an object that has a `path` property, e.g. `callback(filePath)` or `callback({path: filePath})`.

When `callback` is called with nothing, a number, or an object that has an `error` property, the `request` will fail with the `error` number you specified. For the available error numbers you can use, please see the net error list.

By default the `scheme` is treated like `http:`, which is parsed differently than protocols that follow the "generic URI syntax" like `file:`, so you probably want to call `protocol.registerStandardSchemes` to have your scheme treated as a standard scheme.

`protocol.registerBufferProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
  - `request` Object
    - `url` String
    - `referrer` String
    - `method` String
    - `uploadData` UploadData[]
  - `callback` Function
    - `buffer` (Buffer | MimeTypedBuffer) (optional)
- `completion` Function (optional)
  - `error` Error

Registers a protocol of `scheme` that will send a `Buffer` as a response.

The usage is the same with `registerFileProtocol` , except that the `callback` should be called with either a `Buffer` object or an object that has the `data` , `mimeType` , and `charset` properties.

Example:

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

protocol.registerBufferProtocol('atom', (request, callback) => {
  callback({mimeType: 'text/html', data: new Buffer('<h5>Response</h5>')})
}, (error) => {
  if (error) console.error('Failed to register protocol')
})
```

`protocol.registerStringProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
  - `request` Object
    - `url` String
    - `referrer` String
    - `method` String
    - `uploadData` UploadData[]
  - `callback` Function
    - `data` String (optional)
- `completion` Function (optional)
  - `error` Error

Registers a protocol of `scheme` that will send a `String` as a response.

The usage is the same with `registerFileProtocol` , except that the `callback` should be called with either a `String` or an object that has the `data` , `mimeType` , and `charset` properties.

`protocol.registerHttpProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
  - `request` Object
    - `url` String
    - `referrer` String
    - `method` String
    - `uploadData` UploadData[]
  - `callback` Function
    - `redirectRequest` Object
      - `url` String
      - `method` String
      - `session` Object (optional)
      - `uploadData` Object (optional)
        - `contentType` String - MIME type of the content.
        - `data` String - Content to be sent.
- `completion` Function (optional)
  - `error` Error

Registers a protocol of `scheme` that will send an HTTP request as a response.

The usage is the same with `registerFileProtocol` , except that the `callback` should be called with a `redirectRequest` object that has the `url` , `method` , `referrer` , `uploadData` and `session` properties.

By default the HTTP request will reuse the current session. If you want the request to have a different session you should set `session` to `null` .

For POST requests the `uploadData` object must be provided.

`protocol.unregisterProtocol(scheme[, completion])`

- `scheme` String
- `completion` Function (optional)
  - `error` Error

Unregisters the custom protocol of `scheme` .

`protocol.isProtocolHandled(scheme, callback)`

- `scheme` String
- `callback` Function
  - `error` Error

The `callback` will be called with a boolean that indicates whether there is already a handler for `scheme` .

`protocol.interceptFileProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
  - `request` Object
    - `url` String
    - `referrer` String
    - `method` String
    - `uploadData` UploadData[]
  - `callback` Function
    - `filePath` String
- `completion` Function (optional)
  - `error` Error

Intercepts `scheme` protocol and uses `handler` as the protocol's new handler which sends a file as a response.

`protocol.interceptStringProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
  - `request` Object
    - `url` String
    - `referrer` String
    - `method` String
    - `uploadData` UploadData[]
  - `callback` Function
    - `data` String (optional)
- `completion` Function (optional)
  - `error` Error

Intercepts `scheme` protocol and uses `handler` as the protocol's new handler which sends a `String` as a response.

`protocol.interceptBufferProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
  - `request` Object
    - `url` String
    - `referrer` String
    - `method` String
    - `uploadData` UploadData[]
  - `callback` Function
    - `buffer` Buffer (optional)
- `completion` Function (optional)
  - `error` Error

Intercepts `scheme` protocol and uses `handler` as the protocol's new handler which sends a `Buffer` as a response.

`protocol.interceptHttpProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
  - `request` Object
    - `url` String
    - `referrer` String
    - `method` String
    - `uploadData` UploadData[]
  - `callback` Function
    - `redirectRequest` Object
      - `url` String
      - `method` String
      - `session` Object (optional)
      - `uploadData` Object (optional)
        - `contentType` String - MIME type of the content.
        - `data` String - Content to be sent.
- `completion` Function (optional)
  - `error` Error

Intercepts `scheme` protocol and uses `handler` as the protocol's new handler which sends a new HTTP request as a response.

`protocol.uninterceptProtocol(scheme[, completion])`

- `scheme` String
- `completion` Function (optional)
  - `error` Error

Remove the interceptor installed for `scheme` and restore its original handler.

session

> Manage browser sessions, cookies, cache, proxy settings, etc.

Process: Main

The `session` module can be used to create new `Session` objects.

You can also access the `session` of existing pages by using the `session` property of `WebContents`, or from the `session` module.

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

let win = new BrowserWindow({width: 800, height: 600})
win.loadURL('http://github.com')

const ses = win.webContents.session
console.log(ses.getUserAgent())
```

Methods

The `session` module has the following methods:

`session.fromPartition(partition[, options])`

- `partition` String
- `options` Object
  - `cache` Boolean - Whether to enable cache.

Returns `Session` - A session instance from `partition` string. When there is an existing `Session` with the same `partition`, it will be returned; otherwise a new `Session` instance will be created with `options`.

If `partition` starts with `persist:`, the page will use a persistent session available to all pages in the app with the same `partition`. if there is no `persist:` prefix, the page will use an in-memory session. If the `partition` is empty then default session of the app will be returned.

To create a `Session` with `options`, you have to ensure the `Session` with the `partition` has never been used before. There is no way to change the `options` of an existing `Session` object.

Properties

The `session` module has the following properties:

`session.defaultSession`

A `Session` object, the default session object of the app.

Class: Session

> Get and set properties of a session.

Process: Main

You can create a `Session` object in the `session` module:

```
const {session} = require('electron')
const ses = session.fromPartition('persist:name')
console.log(ses.getUserAgent())
```

Instance Events

The following events are available on instances of `Session` :

Event: 'will-download'

- `event` Event
- `item` DownloadItem
- `webContents` WebContents

Emitted when Electron is about to download `item` in `webContents` .

Calling `event.preventDefault()` will cancel the download and `item` will not be available from next tick of the process.

```
const {session} = require('electron')
session.defaultSession.on('will-download', (event, item, webContents) => {
  event.preventDefault()
  require('request')(item.getURL(), (data) => {
    require('fs').writeFileSync('/somewhere', data)
  })
})
```

Instance Methods

The following methods are available on instances of `Session` :

`ses.getCacheSize(callback)`

- `callback` Function
  - `size` Integer - Cache size used in bytes.

Callback is invoked with the session's current cache size.

`ses.clearCache(callback)`

- `callback` Function - Called when operation is done

Clears the session’s HTTP cache.

`ses.clearStorageData([options, callback])`

- `options` Object (optional)
  - `origin` String - Should follow `window.location.origin` ’s representation `scheme://host:port` .
  - `storages` String[] - The types of storages to clear, can contain: `appcache` , `cookies` , `filesystem` , `indexdb` , `localstorage` , `shadercache` , `websql` , `serviceworkers`
  - `quotas` String[] - The types of quotas to clear, can contain: `temporary` , `persistent` , `syncable` .
- `callback` Function (optional) - Called when operation is done.

Clears the data of web storages.

`ses.flushStorageData()`

Writes any unwritten DOMStorage data to disk.

`ses.setProxy(config, callback)`

- `config` Object
  - `pacScript` String - The URL associated with the PAC file.
  - `proxyRules` String - Rules indicating which proxies to use.
  - `proxyBypassRules` String - Rules indicating which URLs should bypass the proxy settings.
- `callback` Function - Called when operation is done.

Sets the proxy settings.

When `pacScript` and `proxyRules` are provided together, the `proxyRules` option is ignored and `pacScript` configuration is applied.

The `proxyRules` has to follow the rules below:

```
proxyRules = schemeProxies[";"<schemeProxies>]
schemeProxies = [<urlScheme>"="]<proxyURIList>
urlScheme = "http" | "https" | "ftp" | "socks"
proxyURIList = <proxyURL>[","<proxyURIList>]
proxyURL = [<proxyScheme>"://"]<proxyHost>[":"<proxyPort>]
```

For example:

- `http=foopy:80;ftp=foopy2` - Use HTTP proxy `foopy:80` for `http://` URLs, and HTTP proxy `foopy2:80` for `ftp://` URLs.
- `foopy:80` - Use HTTP proxy `foopy:80` for all URLs.
- `foopy:80,bar,direct://` - Use HTTP proxy `foopy:80` for all URLs, failing over to `bar` if `foopy:80` is unavailable, and after that using no proxy.
- `socks4://foopy` - Use SOCKS v4 proxy `foopy:1080` for all URLs.
- `http=foopy,socks5://bar.com` - Use HTTP proxy `foopy` for http URLs, and fail over to the SOCKS5 proxy `bar.com` if `foopy` is unavailable.
- `http=foopy,direct://` - Use HTTP proxy `foopy` for http URLs, and use no proxy if `foopy` is unavailable.
- `http=foopy;socks=foopy2` - Use HTTP proxy `foopy` for http URLs, and use `socks4://foopy2` for all other URLs.

The `proxyBypassRules` is a comma separated list of rules described below:

- `[ URL_SCHEME "://" ] HOSTNAME_PATTERN [ ":" <port> ]`

  Match all hostnames that match the pattern HOSTNAME_PATTERN.

  Examples: "foobar.com", "*foobar.com", "*.foobar.com", "*foobar.com:99", "https://x.*.y.com:99"

  - `"." HOSTNAME_SUFFIX_PATTERN [ ":" PORT ]`

    Match a particular domain suffix.

    Examples: ".google.com", ".com", "http://.google.com"

- `[ SCHEME "://" ] IP_LITERAL [ ":" PORT ]`

  Match URLs which are IP address literals.

  Examples: "127.0.1", "[0:0::1]", "[::1]", "http://[::1]:99"

- `IP_LITERAL "/" PREFIX_LENGHT_IN_BITS`

  Match any URL that is to an IP literal that falls between the given range. IP range is specified using CIDR notation.

  Examples: "192.168.1.1/16", "fefe:13::abc/33".

- `<local>`

  Match local addresses. The meaning of `<local>` is whether the host matches one of: "127.0.0.1", "::1", "localhost".

`ses.resolveProxy(url, callback)`

- `url` URL
- `callback` Function
  - `proxy` String

Resolves the proxy information for `url`. The `callback` will be called with `callback(proxy)` when the request is performed.

`ses.setDownloadPath(path)`

- `path` String - The download location

Sets download saving directory. By default, the download directory will be the `Downloads` under the respective app folder.

`ses.enableNetworkEmulation(options)`

- `options` Object
  - `offline` Boolean (optional) - Whether to emulate network outage. Defaults to false.
  - `latency` Double (optional) - RTT in ms. Defaults to 0 which will disable latency throttling.
  - `downloadThroughput` Double (optional) - Download rate in Bps. Defaults to 0 which will disable download throttling.
  - `uploadThroughput` Double (optional) - Upload rate in Bps. Defaults to 0 which will disable upload throttling.

Emulates network with the given configuration for the `session`.

```
// To emulate a GPRS connection with 50kbps throughput and 500 ms latency.
window.webContents.session.enableNetworkEmulation({
  latency: 500,
  downloadThroughput: 6400,
  uploadThroughput: 6400
})

// To emulate a network outage.
window.webContents.session.enableNetworkEmulation({offline: true})
```

`ses.disableNetworkEmulation()`

Disables any network emulation already active for the `session`. Resets to the original network configuration.

`ses.setCertificateVerifyProc(proc)`

- `proc` Function
  - `request` Object
    - `hostname` String
    - `certificate` Certificate
    - `error` String - Verification result from chromium.
  - `callback` Function
    - `verificationResult` Integer - Value can be one of certificate error codes from here. Apart from the certificate error codes, the following special codes can be used.
      - `0` - Indicates success and disables Certificate Transparency verification.
      - `-2` - Indicates failure.
      - `-3` - Uses the verification result from chromium.

Sets the certificate verify proc for `session`, the `proc` will be called with `proc(request, callback)` whenever a server certificate verification is requested. Calling `callback(0)` accepts the certificate, calling `callback(-2)` rejects it.

Calling `setCertificateVerifyProc(null)` will revert back to default certificate verify proc.

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()

win.webContents.session.setCertificateVerifyProc((request, callback) => {
  const {hostname} = request
  if (hostname === 'github.com') {
    callback(0)
  } else {
    callback(-2)
  }
})
```

`ses.setPermissionRequestHandler(handler)`

- `handler` Function
  - `webContents` Object - WebContents requesting the permission.
  - `permission` String - Enum of 'media', 'geolocation', 'notifications', 'midiSysex', 'pointerLock', 'fullscreen', 'openExternal'.
  - `callback` Function
    - `permissionGranted` Boolean - Allow or deny the permission

Sets the handler which can be used to respond to permission requests for the `session`. Calling `callback(true)` will allow the permission and `callback(false)` will reject it.

```
const {session} = require('electron')
session.fromPartition('some-partition').setPermissionRequestHandler((webContents, permission, callback) => {
  if (webContents.getURL() === 'some-host' && permission === 'notifications') {
    return callback(false) // denied.
  }

  callback(true)
})
```

`ses.clearHostResolverCache([callback])`

- `callback` Function (optional) - Called when operation is done.

Clears the host resolver cache.

`ses.allowNTLMCredentialsForDomains(domains)`

- `domains` String - A comma-seperated list of servers for which integrated authentication is enabled.

Dynamically sets whether to always send credentials for HTTP NTLM or Negotiate authentication.

```
const {session} = require('electron')
// consider any url ending with `example.com`, `foobar.com`, `baz`
// for integrated authentication.
session.defaultSession.allowNTLMCredentialsForDomains('*example.com, *foobar.com, *baz')

// consider all urls for integrated authentication.
session.defaultSession.allowNTLMCredentialsForDomains('*')
```

`ses.setUserAgent(userAgent[, acceptLanguages])`

- `userAgent` String
- `acceptLanguages` String (optional)

Overrides the `userAgent` and `acceptLanguages` for this session.

The `acceptLanguages` must a comma separated ordered list of language codes, for example `"en-US,fr,de,ko,zh-CN,ja"`.

This doesn't affect existing `WebContents`, and each `WebContents` can use `webContents.setUserAgent` to override the session-wide user agent.

`ses.getUserAgent()`

Returns `String` - The user agent for this session.

`ses.getBlobData(identifier, callback)`

- `identifier` String - Valid UUID.
- `callback` Function
  - `result` Buffer - Blob data.

Returns `Blob` - The blob data associated with the `identifier`.

`ses.createInterruptedDownload(options)`

- `options` Object
  - `path` String - Absolute path of the download.
  - `urlChain` String[] - Complete URL chain for the download.
  - `mimeType` String (optional)
  - `offset` Integer - Start range for the download.
  - `length` Integer - Total length of the download.
  - `lastModified` String - Last-Modified header value.
  - `eTag` String - ETag header value.
  - `startTime` Double (optional) - Time when download was started in number of seconds since UNIX epoch.

Allows resuming `cancelled` or `interrupted` downloads from previous `Session`. The API will generate a DownloadItem that can be accessed with the will-download event. The DownloadItem will not have any `WebContents` associated with it and the initial state will be `interrupted`. The download will start only when the `resume` API is called on the DownloadItem.

`ses.clearAuthCache(options[, callback])`

- `options` (RemovePassword | RemoveClientCertificate)
- `callback` Function (optional) - Called when operation is done

Clears the session’s HTTP authentication cache.

Instance Properties

The following properties are available on instances of `Session`:

`ses.cookies`

A Cookies object for this session.

`ses.webRequest`

A WebRequest object for this session.

`ses.protocol`

A Protocol object (an instance of protocol module) for this session.

```
const {app, session} = require('electron')
const path = require('path')

app.on('ready', function () {
  const protocol = session.fromPartition('some-partition').protocol
  protocol.registerFileProtocol('atom', function (request, callback) {
    var url = request.url.substr(7)
    callback({path: path.normalize(`${__dirname}/${url}`)})
  }, function (error) {
    if (error) console.error('Failed to register protocol')
  })
})
```

systemPreferences

> Get system preferences.

Process: Main

```
const {systemPreferences} = require('electron')
console.log(systemPreferences.isDarkMode())
```

Events

The `systemPreferences` object emits the following events:

Event: 'accent-color-changed' *Windows*

Returns:

- `event` Event
- `newColor` String - The new RGBA color the user assigned to be their system accent color.

Event: 'color-changed' *Windows*

Returns:

- `event` Event

Event: 'inverted-color-scheme-changed' *Windows*

Returns:

- `event` Event
- `invertedColorScheme` Boolean - `true` if an inverted color scheme, such as a high contrast theme, is being used, `false` otherwise.

Methods

`systemPreferences.isDarkMode()` *macOS*

Returns `Boolean` - Whether the system is in Dark Mode.

`systemPreferences.isSwipeTrackingFromScrollEventsEnabled()` *macOS*

Returns `Boolean` - Whether the Swipe between pages setting is on.

`systemPreferences.postNotification(event, userInfo)` *macOS*

- `event` String
- `userInfo` Object

Posts `event` as native notifications of macOS. The `userInfo` is an Object that contains the user information dictionary sent along with the notification.

`systemPreferences.postLocalNotification(event, userInfo)` *macOS*

- `event` String
- `userInfo` Object

Posts `event` as native notifications of macOS. The `userInfo` is an Object that contains the user information dictionary sent along with the notification.

`systemPreferences.subscribeNotification(event, callback)` *macOS*

- `event` String
- `callback` Function
  - `event` String
  - `userInfo` Object

Subscribes to native notifications of macOS, `callback` will be called with `callback(event, userInfo)` when the corresponding `event` happens. The `userInfo` is an Object that contains the user information dictionary sent along with the notification.

The `id` of the subscriber is returned, which can be used to unsubscribe the `event` .

Under the hood this API subscribes to `NSDistributedNotificationCenter` , example values of `event` are:

- `AppleInterfaceThemeChangedNotification`
- `AppleAquaColorVariantChanged`
- `AppleColorPreferencesChangedNotification`
- `AppleShowScrollBarsSettingChanged`

`systemPreferences.unsubscribeNotification(id)` *macOS*

- `id` Integer

Removes the subscriber with `id` .

`systemPreferences.subscribeLocalNotification(event, callback)` *macOS*

- `event` String
- `callback` Function
  - `event` String
  - `userInfo` Object

Same as `subscribeNotification` , but uses `NSNotificationCenter` for local defaults. This is necessary for events such as `NSUserDefaultsDidChangeNotification`

`systemPreferences.unsubscribeLocalNotification(id)` *macOS*

- `id` Integer

Same as `unsubscribeNotification` , but removes the subscriber from `NSNotificationCenter` .

`systemPreferences.getUserDefault(key, type)` *macOS*

- `key` String
- `type` String - Can be `string` , `boolean` , `integer` , `float` , `double` , `url` , `array` , `dictionary`

Get the value of `key` in system preferences.

This API uses `NSUserDefaults` on macOS. Some popular `key` and `type` s are:

- `AppleInterfaceStyle` : `string`
- `AppleAquaColorVariant` : `integer`
- `AppleHighlightColor` : `string`
- `AppleShowScrollBars` : `string`
- `NSNavRecentPlaces` : `array`
- `NSPreferredWebServices` : `dictionary`
- `NSUserDictionaryReplacementItems` : `array`

`systemPreferences.setUserDefault(key, type, value)` *macOS*

- `key` String
- `type` String - See [`getUserDefault`][#systempreferencesgetuserdefaultkey-type-macos]
- `value` String

Set the value of `key` in system preferences.

Note that `type` should match actual type of `value` . An exception is thrown if they don't.

This API uses `NSUserDefaults` on macOS. Some popular `key` and `type` s are:

- `ApplePressAndHoldEnabled` : `boolean`

`systemPreferences.isAeroGlassEnabled()` *Windows*

This method returns `true` if DWM composition (Aero Glass) is enabled, and `false` otherwise.

An example of using it to determine if you should create a transparent window or not (transparent windows won't work correctly when DWM composition is disabled):

```
const {BrowserWindow, systemPreferences} = require('electron')
let browserOptions = {width: 1000, height: 800}

// Make the window transparent only if the platform supports it.
if (process.platform !== 'win32' || systemPreferences.isAeroGlassEnabled()) {
  browserOptions.transparent = true
  browserOptions.frame = false
}

// Create the window.
let win = new BrowserWindow(browserOptions)

// Navigate.
if (browserOptions.transparent) {
  win.loadURL(`file://${__dirname}/index.html`)
} else {
  // No transparency, so we load a fallback that uses basic styles.
  win.loadURL(`file://${__dirname}/fallback.html`)
}
```

`systemPreferences.getAccentColor()` *Windows*

Returns `String` - The users current system wide accent color preference in RGBA hexadecimal form.

```
const color = systemPreferences.getAccentColor() // `"aabbccdd"`
const red = color.substr(0, 2) // "aa"
const green = color.substr(2, 2) // "bb"
const blue = color.substr(4, 2) // "cc"
const alpha = color.substr(6, 2) // "dd"
```

`systemPreferences.getColor(color)` *Windows*

- `color` String - One of the following values:
  - `3d-dark-shadow` - Dark shadow for three-dimensional display elements.
  - `3d-face` - Face color for three-dimensional display elements and for dialog box backgrounds.
  - `3d-highlight` - Highlight color for three-dimensional display elements.
  - `3d-light` - Light color for three-dimensional display elements.
  - `3d-shadow` - Shadow color for three-dimensional display elements.
  - `active-border` - Active window border.
  - `active-caption` - Active window title bar. Specifies the left side color in the color gradient of an active window's title bar if the gradient effect is enabled.
  - `active-caption-gradient` - Right side color in the color gradient of an active window's title bar.
  - `app-workspace` - Background color of multiple document interface (MDI) applications.
  - `button-text` - Text on push buttons.
  - `caption-text` - Text in caption, size box, and scroll bar arrow box.
  - `desktop` - Desktop background color.
  - `disabled-text` - Grayed (disabled) text.
  - `highlight` - Item(s) selected in a control.
  - `highlight-text` - Text of item(s) selected in a control.
  - `hotlight` - Color for a hyperlink or hot-tracked item.
  - `inactive-border` - Inactive window border.
  - `inactive-caption` - Inactive window caption. Specifies the left side color in the color gradient of an inactive window's title bar if the gradient effect is enabled.
  - `inactive-caption-gradient` - Right side color in the color gradient of an inactive window's title bar.
  - `inactive-caption-text` - Color of text in an inactive caption.
  - `info-background` - Background color for tooltip controls.
  - `info-text` - Text color for tooltip controls.
  - `menu` - Menu background.
  - `menu-highlight` - The color used to highlight menu items when the menu appears as a flat menu.
  - `menubar` - The background color for the menu bar when menus appear as flat menus.
  - `menu-text` - Text in menus.
  - `scrollbar` - Scroll bar gray area.
  - `window` - Window background.
  - `window-frame` - Window frame.
  - `window-text` - Text in windows.

Returns `String` - The system color setting in RGB hexadecimal form (`#ABCDEF`). See the Windows docs for more details.

`systemPreferences.isInvertedColorScheme()` *Windows*

Returns `Boolean` - `true` if an inverted color scheme, such as a high contrast theme, is active, `false` otherwise.

Class: Tray

> Add icons and context menus to the system's notification area.

Process: Main

`Tray` is an EventEmitter.

```
const {app, Menu, Tray} = require('electron')

let tray = null
app.on('ready', () => {
  tray = new Tray('/path/to/my/icon')
  const contextMenu = Menu.buildFromTemplate([
    {label: 'Item1', type: 'radio'},
    {label: 'Item2', type: 'radio'},
    {label: 'Item3', type: 'radio', checked: true},
    {label: 'Item4', type: 'radio'}
  ])
  tray.setToolTip('This is my application.')
  tray.setContextMenu(contextMenu)
})
```

Platform limitations:

- On Linux the app indicator will be used if it is supported, otherwise `GtkStatusIcon` will be used instead.
- On Linux distributions that only have app indicator support, you have to install `libappindicator1` to make the tray icon work.
- App indicator will only be shown when it has a context menu.
- When app indicator is used on Linux, the `click` event is ignored.
- On Linux in order for changes made to individual `MenuItem`s to take effect, you have to call `setContextMenu` again. For example:

```
const {app, Menu, Tray} = require('electron')

let appIcon = null
app.on('ready', () => {
  appIcon = new Tray('/path/to/my/icon')
  const contextMenu = Menu.buildFromTemplate([
    {label: 'Item1', type: 'radio'},
    {label: 'Item2', type: 'radio'}
  ])

  // Make a change to the context menu
  contextMenu.items[1].checked = false

  // Call this again for Linux because we modified the context menu
  appIcon.setContextMenu(contextMenu)
})
```

- On Windows it is recommended to use `ICO` icons to get best visual effects.

If you want to keep exact same behaviors on all platforms, you should not rely on the `click` event and always attach a context menu to the tray icon.

`new Tray(image)`

- `image` (NativeImage | String)

Creates a new tray icon associated with the `image`.

Instance Events

The `Tray` module emits the following events:

Event: 'click'

- `event` Event
  - `altKey` Boolean
  - `shiftKey` Boolean
  - `ctrlKey` Boolean
  - `metaKey` Boolean
- `bounds` Rectangle - The bounds of tray icon

Emitted when the tray icon is clicked.

Event: 'right-click' *macOS Windows*

- `event` Event
  - `altKey` Boolean
  - `shiftKey` Boolean
  - `ctrlKey` Boolean
  - `metaKey` Boolean
- `bounds` Rectangle - The bounds of tray icon

Emitted when the tray icon is right clicked.

Event: 'double-click' *macOS Windows*

- `event` Event
  - `altKey` Boolean
  - `shiftKey` Boolean
  - `ctrlKey` Boolean
  - `metaKey` Boolean
- `bounds` Rectangle - The bounds of tray icon

Emitted when the tray icon is double clicked.

Event: 'balloon-show' *Windows*

Emitted when the tray balloon shows.

Event: 'balloon-click' *Windows*

Emitted when the tray balloon is clicked.

Event: 'balloon-closed' *Windows*

Emitted when the tray balloon is closed because of timeout or user manually closes it.

Event: 'drop' *macOS*

Emitted when any dragged items are dropped on the tray icon.

Event: 'drop-files' *macOS*

- `event` Event

- `files` String[] - The paths of the dropped files.

Emitted when dragged files are dropped in the tray icon.

Event: 'drop-text' *macOS*

- `event` Event
- `text` String - the dropped text string

Emitted when dragged text is dropped in the tray icon.

Event: 'drag-enter' *macOS*

Emitted when a drag operation enters the tray icon.

Event: 'drag-leave' *macOS*

Emitted when a drag operation exits the tray icon.

Event: 'drag-end' *macOS*

Emitted when a drag operation ends on the tray or ends at another location.

Instance Methods

The `Tray` class has the following methods:

`tray.destroy()`

Destroys the tray icon immediately.

`tray.setImage(image)`

- `image` (NativeImage | String)

Sets the `image` associated with this tray icon.

`tray.setPressedImage(image)` *macOS*

- `image` NativeImage

Sets the `image` associated with this tray icon when pressed on macOS.

`tray.setToolTip(toolTip)`

- `toolTip` String

Sets the hover text for this tray icon.

`tray.setTitle(title)` *macOS*

- `title` String

Sets the title displayed aside of the tray icon in the status bar.

`tray.setHighlightMode(mode)` *macOS*

- `mode` String - Highlight mode with one of the following values:

  - `selection` - Highlight the tray icon when it is clicked and also when its context menu is open. This is the default.
  - `always` - Always highlight the tray icon.
  - `never` - Never highlight the tray icon.

Sets when the tray's icon background becomes highlighted (in blue).

Note: You can use `highlightMode` with a `BrowserWindow` by toggling between `'never'` and `'always'` modes when the window visibility changes.

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

const win = new BrowserWindow({width: 800, height: 600})
const tray = new Tray('/path/to/my/icon')

tray.on('click', () => {
  win.isVisible() ? win.hide() : win.show()
})
win.on('show', () => {
  tray.setHighlightMode('always')
})
win.on('hide', () => {
  tray.setHighlightMode('never')
})
```

`tray.displayBalloon(options)` *Windows*

- `options` Object
  - `icon` (NativeImage | String) - (optional)
  - `title` String - (optional)
  - `content` String - (optional)

Displays a tray balloon.

`tray.popUpContextMenu([menu, position])` *macOS Windows*

- `menu` Menu (optional)
- `position` Object (optional) - The pop up position.
  - `x` Integer
  - `y` Integer

Pops up the context menu of the tray icon. When `menu` is passed, the `menu` will be shown instead of the tray icon's context menu.

The `position` is only available on Windows, and it is (0, 0) by default.

`tray.setContextMenu(menu)`

- `menu` Menu

Sets the context menu for this icon.

`tray.getBounds()` *macOS Windows*

Returns `Rectangle`

The `bounds` of this tray icon as `Object`.

`tray.isDestroyed()`

Returns `Boolean` - Whether the tray icon is destroyed.

webContents

> Render and control web pages.

Process: Main

`webContents` is an EventEmitter. It is responsible for rendering and controlling a web page and is a property of the `BrowserWindow` object. An example of accessing the `webContents` object:

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

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

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

Methods

These methods can be accessed from the `webContents` module:

```
const {webContents} = require('electron')
console.log(webContents)
```

`webContents.getAllWebContents()`

Returns `WebContents[]` - An array of all `WebContents` instances. This will contain web contents for all windows, webviews, opened devtools, and devtools extension background pages.

`webContents.getFocusedWebContents()`

Returns `WebContents` - The web contents that is focused in this application, otherwise returns `null`.

`webContents.fromId(id)`

- `id` Integer

Returns `WebContents` - A WebContents instance with the given ID.

Class: WebContents

> Render and control the contents of a BrowserWindow instance.

Process: Main

Instance Events

Event: 'did-finish-load'

Emitted when the navigation is done, i.e. the spinner of the tab has stopped spinning, and the `onload` event was dispatched.

Event: 'did-fail-load'

Returns:

- `event` Event
- `errorCode` Integer
- `errorDescription` String
- `validatedURL` String
- `isMainFrame` Boolean

This event is like `did-finish-load` but emitted when the load failed or was cancelled, e.g. `window.stop()` is invoked. The full list of error codes and their meaning is available here.

Event: 'did-frame-finish-load'

Returns:

- `event` Event
- `isMainFrame` Boolean

Emitted when a frame has done navigation.

Event: 'did-start-loading'

Corresponds to the points in time when the spinner of the tab started spinning.

Event: 'did-stop-loading'

Corresponds to the points in time when the spinner of the tab stopped spinning.

Event: 'did-get-response-details'

Returns:

- `event` Event
- `status` Boolean
- `newURL` String
- `originalURL` String
- `httpResponseCode` Integer
- `requestMethod` String
- `referrer` String
- `headers` Object
- `resourceType` String

Emitted when details regarding a requested resource are available. `status` indicates the socket connection to download the resource.

Event: 'did-get-redirect-request'

Returns:

- `event` Event
- `oldURL` String
- `newURL` String
- `isMainFrame` Boolean
- `httpResponseCode` Integer
- `requestMethod` String
- `referrer` String
- `headers` Object

Emitted when a redirect is received while requesting a resource.

Event: 'dom-ready'

Returns:

- `event` Event

Emitted when the document in the given frame is loaded.

Event: 'page-favicon-updated'

Returns:

- `event` Event
- `favicons` String[] - Array of URLs

Emitted when page receives favicon urls.

Event: 'new-window'

Returns:

- `event` Event
- `url` String
- `frameName` String
- `disposition` String - Can be `default` , `foreground-tab` , `background-tab` , `new-window` , `save-to-disk` and `other` .
- `options` Object - The options which will be used for creating the new `BrowserWindow` .
- `additionalFeatures` String[] - The non-standard features (features not handled by Chromium or Electron) given to `window.open()` .

Emitted when the page requests to open a new window for a `url` . It could be requested by `window.open` or an external link like `<a target='_blank'>` .

By default a new `BrowserWindow` will be created for the `url` .

Calling `event.preventDefault()` will prevent Electron from automatically creating a new `BrowserWindow` . If you call `event.preventDefault()` and manually create a new `BrowserWindow` then you must set `event.newGuest` to reference the new `BrowserWindow` instance, failing to do so may result in unexpected behavior. For example:

```
myBrowserWindow.webContents.on('new-window', (event, url) => {
  event.preventDefault()
  const win = new BrowserWindow({show: false})
  win.once('ready-to-show', () => win.show())
  win.loadURL(url)
  event.newGuest = win
})
```

Event: 'will-navigate'

Returns:

- `event` Event
- `url` String

Emitted when a user or the page wants to start navigation. It can happen when the `window.location` object is changed or a user clicks a link in the page.

This event will not emit when the navigation is started programmatically with APIs like `webContents.loadURL` and `webContents.back` .

It is also not emitted for in-page navigations, such as clicking anchor links or updating the `window.location.hash`. Use `did-navigate-in-page` event for this purpose.

Calling `event.preventDefault()` will prevent the navigation.

Event: 'did-navigate'

Returns:

- `event` Event
- `url` String

Emitted when a navigation is done.

This event is not emitted for in-page navigations, such as clicking anchor links or updating the `window.location.hash`. Use `did-navigate-in-page` event for this purpose.

Event: 'did-navigate-in-page'

Returns:

- `event` Event
- `url` String
- `isMainFrame` Boolean

Emitted when an in-page navigation happened.

When in-page navigation happens, the page URL changes but does not cause navigation outside of the page. Examples of this occurring are when anchor links are clicked or when the DOM `hashchange` event is triggered.

Event: 'crashed'

Returns:

- `event` Event
- `killed` Boolean

Emitted when the renderer process crashes or is killed.

Event: 'plugin-crashed'

Returns:

- `event` Event
- `name` String
- `version` String

Emitted when a plugin process has crashed.

Event: 'destroyed'

Emitted when `webContents` is destroyed.

Event: 'before-input-event'

Returns:

- `event` Event
- `input` Object - Input properties
    - `type` String - Either `keyUp` or `keyDown`

    - `key` String - Equivalent to KeyboardEvent.key
    - `code` String - Equivalent to KeyboardEvent.code
    - `isAutoRepeat` Boolean - Equivalent to KeyboardEvent.repeat
    - `shift` Boolean - Equivalent to KeyboardEvent.shiftKey
    - `control` Boolean - Equivalent to KeyboardEvent.controlKey
    - `alt` Boolean - Equivalent to KeyboardEvent.altKey
    - `meta` Boolean - Equivalent to KeyboardEvent.metaKey

Emitted before dispatching the `keydown` and `keyup` events in the page. Calling `event.preventDefault` will prevent the page `keydown` / `keyup` events from being dispatched.

Event: 'devtools-opened'

Emitted when DevTools is opened.

Event: 'devtools-closed'

Emitted when DevTools is closed.

Event: 'devtools-focused'

Emitted when DevTools is focused / opened.

Event: 'certificate-error'

Returns:

- `event` Event
- `url` String
- `error` String - The error code
- `certificate` Certificate
- `callback` Function
    - `isTrusted` Boolean - Indicates whether the certificate can be considered trusted

Emitted when failed to verify the `certificate` for `url`.

The usage is the same with the `certificate-error` event of `app`.

Event: 'select-client-certificate'

Returns:

- `event` Event
- `url` URL
- `certificateList` Certificate[]
- `callback` Function
    - `certificate` Certificate - Must be a certificate from the given list

Emitted when a client certificate is requested.

The usage is the same with the `select-client-certificate` event of `app`.

Event: 'login'

Returns:

- `event` Event
- `request` Object

    - `method` String
    - `url` URL
    - `referrer` URL
- `authInfo` Object
    - `isProxy` Boolean
    - `scheme` String
    - `host` String
    - `port` Integer
    - `realm` String
- `callback` Function
    - `username` String
    - `password` String

Emitted when `webContents` wants to do basic auth.

The usage is the same with the `login` event of `app` .

Event: 'found-in-page'

Returns:

- `event` Event
- `result` Object
    - `requestId` Integer
    - `activeMatchOrdinal` Integer - Position of the active match.
    - `matches` Integer - Number of Matches.
    - `selectionArea` Object - Coordinates of first match region.

Emitted when a result is available for [`webContents.findInPage`] request.

Event: 'media-started-playing'

Emitted when media starts playing.

Event: 'media-paused'

Emitted when media is paused or done playing.

Event: 'did-change-theme-color'

Emitted when a page's theme color changes. This is usually due to encountering a meta tag:

```
<meta name='theme-color' content='#ff0000'>
```

Event: 'update-target-url'

Returns:

- `event` Event
- `url` String

Emitted when mouse moves over a link or the keyboard moves the focus to a link.

Event: 'cursor-changed'

Returns:

- `event` Event
- `type` String
- `image` NativeImage (optional)
- `scale` Float (optional) - scaling factor for the custom cursor
- `size` Object (optional) - the size of the `image`
  - `width` Integer
  - `height` Integer
- `hotspot` Object (optional) - coordinates of the custom cursor's hotspot
  - `x` Integer - x coordinate
  - `y` Integer - y coordinate

Emitted when the cursor's type changes. The `type` parameter can be `default` , `crosshair` , `pointer` , `text` , `wait` , `help` , `e-resize` , `n-resize` , `ne-resize` , `nw-resize` , `s-resize` , `se-resize` , `sw-resize` , `w-resize` , `ns-resize` , `ew-resize` , `nesw-resize` , `nwse-resize` , `col-resize` , `row-resize` , `m-panning` , `e-panning` , `n-panning` , `ne-panning` , `nw-panning` , `s-panning` , `se-panning` , `sw-panning` , `w-panning` , `move` , `vertical-text` , `cell` , `context-menu` , `alias` , `progress` , `nodrop` , `copy` , `none` , `not-allowed` , `zoom-in` , `zoom-out` , `grab` , `grabbing` , `custom` .

If the `type` parameter is `custom` , the `image` parameter will hold the custom cursor image in a `NativeImage` , and `scale` , `size` and `hotspot` will hold additional information about the custom cursor.

Event: 'context-menu'

Returns:

- `event` Event
- `params` Object
  - `x` Integer - x coordinate
  - `y` Integer - y coordinate
  - `linkURL` String - URL of the link that encloses the node the context menu was invoked on.
  - `linkText` String - Text associated with the link. May be an empty string if the contents of the link are an image.
  - `pageURL` String - URL of the top level page that the context menu was invoked on.
  - `frameURL` String - URL of the subframe that the context menu was invoked on.
  - `srcURL` String - Source URL for the element that the context menu was invoked on. Elements with source URLs are images, audio and video.
  - `mediaType` String - Type of the node the context menu was invoked on. Can be `none` , `image` , `audio` , `video` , `canvas` , `file` or `plugin` .
  - `hasImageContents` Boolean - Whether the context menu was invoked on an image which has non-empty contents.
  - `isEditable` Boolean - Whether the context is editable.
  - `selectionText` String - Text of the selection that the context menu was invoked on.
  - `titleText` String - Title or alt text of the selection that the context was invoked on.
  - `misspelledWord` String - The misspelled word under the cursor, if any.
  - `frameCharset` String - The character encoding of the frame on which the menu was invoked.
  - `inputFieldType` String - If the context menu was invoked on an input field, the type of that field. Possible values are `none` , `plainText` , `password` , `other` .
  - `menuSourceType` String - Input source that invoked the context menu. Can be `none` , `mouse` , `keyboard` , `touch` , `touchMenu` .
  - `mediaFlags` Object - The flags for the media element the context menu was invoked on.
    - `inError` Boolean - Whether the media element has crashed.
    - `isPaused` Boolean - Whether the media element is paused.
    - `isMuted` Boolean - Whether the media element is muted.
    - `hasAudio` Boolean - Whether the media element has audio.
    - `isLooping` Boolean - Whether the media element is looping.
    - `isControlsVisible` Boolean - Whether the media element's controls are visible.
    - `canToggleControls` Boolean - Whether the media element's controls are toggleable.
    - `canRotate` Boolean - Whether the media element can be rotated.

    - `editFlags` Object - These flags indicate whether the renderer believes it is able to perform the corresponding action.
        - `canUndo` Boolean - Whether the renderer believes it can undo.
        - `canRedo` Boolean - Whether the renderer believes it can redo.
        - `canCut` Boolean - Whether the renderer believes it can cut.
        - `canCopy` Boolean - Whether the renderer believes it can copy
        - `canPaste` Boolean - Whether the renderer believes it can paste.
        - `canDelete` Boolean - Whether the renderer believes it can delete.
        - `canSelectAll` Boolean - Whether the renderer believes it can select all.

Emitted when there is a new context menu that needs to be handled.

Event: 'select-bluetooth-device'

Returns:

- `event` Event
- `devices` BluetoothDevice[]
- `callback` Function
    - `deviceId` String

Emitted when bluetooth device needs to be selected on call to `navigator.bluetooth.requestDevice`. To use `navigator.bluetooth` api `webBluetooth` should be enabled. If `event.preventDefault` is not called, first available device will be selected. `callback` should be called with `deviceId` to be selected, passing empty string to `callback` will cancel the request.

```
const {app, webContents} = require('electron')
app.commandLine.appendSwitch('enable-web-bluetooth')

app.on('ready', () => {
  webContents.on('select-bluetooth-device', (event, deviceList, callback) => {
    event.preventDefault()
    let result = deviceList.find((device) => {
      return device.deviceName === 'test'
    })
    if (!result) {
      callback('')
    } else {
      callback(result.deviceId)
    }
  })
})
```

Event: 'paint'

Returns:

- `event` Event
- `dirtyRect` Rectangle
- `image` NativeImage - The image data of the whole frame.

Emitted when a new frame is generated. Only the dirty area is passed in the buffer.

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

let win = new BrowserWindow({webPreferences: {offscreen: true}})
win.webContents.on('paint', (event, dirty, image) => {
  // updateBitmap(dirty, image.getBitmap())
})
win.loadURL('http://github.com')
```

Event: 'devtools-reload-page'

Emitted when the devtools window instructs the webContents to reload

Event: 'will-attach-webview'

Returns:

- `event` Event
- `webPreferences` Object - The web preferences that will be used by the guest page. This object can be modified to adjust the preferences for the guest page.
- `params` Object - The other `<webview>` parameters such as the `src` URL. This object can be modified to adjust the parameters of the guest page.

Emitted when a `<webview>` 's web contents is being attached to this web contents. Calling `event.preventDefault()` will destroy the guest page.

This event can be used to configure `webPreferences` for the `webContents` of a `<webview>` before it's loaded, and provides the ability to set settings that can't be set via `<webview>` attributes.

Instance Methods

`contents.loadURL(url[, options])`

- `url` String
- `options` Object (optional)
  - `httpReferrer` String (optional) - A HTTP Referrer url.
  - `userAgent` String (optional) - A user agent originating the request.
  - `extraHeaders` String (optional) - Extra headers separated by "\n"
  - `postData` (UploadRawData | UploadFile | UploadFileSystem | UploadBlob)[] - (optional)
  - `baseURLForDataURL` String (optional) - Base url (with trailing path separator) for files to be loaded by the data url. This is needed only if the specified `url` is a data url and needs to load other files.

Loads the `url` in the window. The `url` must contain the protocol prefix, e.g. the `http://` or `file://` . If the load should bypass http cache then use the `pragma` header to achieve it.

```
const {webContents} = require('electron')
const options = {extraHeaders: 'pragma: no-cache\n'}
webContents.loadURL('https://github.com', options)
```

`contents.downloadURL(url)`

- `url` String

Initiates a download of the resource at `url` without navigating. The `will-download` event of `session` will be triggered.

`contents.getURL()`

Returns `String` - The URL of the current web page.

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow({width: 800, height: 600})
win.loadURL('http://github.com')

let currentURL = win.webContents.getURL()
console.log(currentURL)
```

`contents.getTitle()`

Returns `String` - The title of the current web page.

`contents.isDestroyed()`

Returns `Boolean` - Whether the web page is destroyed.

`contents.isFocused()`

Returns `Boolean` - Whether the web page is focused.

`contents.isLoading()`

Returns `Boolean` - Whether web page is still loading resources.

`contents.isLoadingMainFrame()`

Returns `Boolean` - Whether the main frame (and not just iframes or frames within it) is still loading.

`contents.isWaitingForResponse()`

Returns `Boolean` - Whether the web page is waiting for a first-response from the main resource of the page.

`contents.stop()`

Stops any pending navigation.

`contents.reload()`

Reloads the current web page.

`contents.reloadIgnoringCache()`

Reloads current page and ignores cache.

`contents.canGoBack()`

Returns `Boolean` - Whether the browser can go back to previous web page.

`contents.canGoForward()`

Returns `Boolean` - Whether the browser can go forward to next web page.

`contents.canGoToOffset(offset)`

- `offset` Integer

Returns `Boolean` - Whether the web page can go to `offset` .

`contents.clearHistory()`

Clears the navigation history.

`contents.goBack()`

Makes the browser go back a web page.

`contents.goForward()`

Makes the browser go forward a web page.

`contents.goToIndex(index)`

- `index` Integer

Navigates browser to the specified absolute web page index.

`contents.goToOffset(offset)`

- `offset` Integer

Navigates to the specified offset from the "current entry".

`contents.isCrashed()`

Returns `Boolean` - Whether the renderer process has crashed.

`contents.setUserAgent(userAgent)`

- `userAgent` String

Overrides the user agent for this web page.

`contents.getUserAgent()`

Returns `String` - The user agent for this web page.

`contents.insertCSS(css)`

- `css` String

Injects CSS into the current web page.

`contents.executeJavaScript(code[, userGesture, callback])`

- `code` String
- `userGesture` Boolean (optional) - Default is `false`.
- `callback` Function (optional) - Called after script has been executed.
    - `result` Any

Returns `Promise` - A promise that resolves with the result of the executed code or is rejected if the result of the code is a rejected promise.

Evaluates `code` in page.

In the browser window some HTML APIs like `requestFullScreen` can only be invoked by a gesture from the user. Setting `userGesture` to `true` will remove this limitation.

If the result of the executed code is a promise the callback result will be the resolved value of the promise. We recommend that you use the returned Promise to handle code that results in a Promise.

```
contents.executeJavaScript('fetch("https://jsonplaceholder.typicode.com/users/1").then(resp => resp.json())', true)
  .then((result) => {
    console.log(result) // Will be the JSON object from the fetch call
  })
```

`contents.setAudioMuted(muted)`

- `muted` Boolean

Mute the audio on the current web page.

`contents.isAudioMuted()`

Returns `Boolean` - Whether this page has been muted.

`contents.setZoomFactor(factor)`

- `factor` Number - Zoom factor.

Changes the zoom factor to the specified factor. Zoom factor is zoom percent divided by 100, so 300% = 3.0.

`contents.getZoomFactor(callback)`

- `callback` Function
  - `zoomFactor` Number

Sends a request to get current zoom factor, the `callback` will be called with `callback(zoomFactor)` .

`contents.setZoomLevel(level)`

- `level` Number - Zoom level

Changes the zoom level to the specified level. The original size is 0 and each increment above or below represents zooming 20% larger or smaller to default limits of 300% and 50% of original size, respectively.

`contents.getZoomLevel(callback)`

- `callback` Function
  - `zoomLevel` Number

Sends a request to get current zoom level, the `callback` will be called with `callback(zoomLevel)` .

`contents.setZoomLevelLimits(minimumLevel, maximumLevel)`

- `minimumLevel` Number
- `maximumLevel` Number

Deprecated: Call `setVisualZoomLevelLimits` instead to set the visual zoom level limits. This method will be removed in Electron 2.0.

`contents.setVisualZoomLevelLimits(minimumLevel, maximumLevel)`

- `minimumLevel` Number
- `maximumLevel` Number

Sets the maximum and minimum pinch-to-zoom level.

`contents.setLayoutZoomLevelLimits(minimumLevel, maximumLevel)`

- `minimumLevel` Number
- `maximumLevel` Number

Sets the maximum and minimum layout-based (i.e. non-visual) zoom level.

`contents.undo()`

Executes the editing command `undo` in web page.

`contents.redo()`

Executes the editing command `redo` in web page.

`contents.cut()`

Executes the editing command `cut` in web page.

`contents.copy()`

Executes the editing command `copy` in web page.

`contents.copyImageAt(x, y)`

- `x` Integer
- `y` Integer

Copy the image at the given position to the clipboard.

`contents.paste()`

Executes the editing command `paste` in web page.

`contents.pasteAndMatchStyle()`

Executes the editing command `pasteAndMatchStyle` in web page.

`contents.delete()`

Executes the editing command `delete` in web page.

`contents.selectAll()`

Executes the editing command `selectAll` in web page.

`contents.unselect()`

Executes the editing command `unselect` in web page.

`contents.replace(text)`

- `text` String

Executes the editing command `replace` in web page.

`contents.replaceMisspelling(text)`

- `text` String

Executes the editing command `replaceMisspelling` in web page.

`contents.insertText(text)`

- `text` String

Inserts `text` to the focused element.

`contents.findInPage(text[, options])`

- `text` String - Content to be searched, must not be empty.
- `options` Object (optional)
  - `forward` Boolean - (optional) Whether to search forward or backward, defaults to `true` .
  - `findNext` Boolean - (optional) Whether the operation is first request or a follow up, defaults to `false` .

    - `matchCase` Boolean - (optional) Whether search should be case-sensitive, defaults to `false` .
    - `wordStart` Boolean - (optional) Whether to look only at the start of words. defaults to `false` .
    - `medialCapitalAsWordStart` Boolean - (optional) When combined with `wordStart` , accepts a match in the middle of a word if the match begins with an uppercase letter followed by a lowercase or non-letter. Accepts several other intra-word matches, defaults to `false` .

Starts a request to find all matches for the `text` in the web page and returns an `Integer` representing the request id used for the request. The result of the request can be obtained by subscribing to `found-in-page` event.

`contents.stopFindInPage(action)`

- `action` String - Specifies the action to take place when ending [`webContents.findInPage`] request.
    - `clearSelection` - Clear the selection.
    - `keepSelection` - Translate the selection into a normal selection.
    - `activateSelection` - Focus and click the selection node.

Stops any `findInPage` request for the `webContents` with the provided `action` .

```
const {webContents} = require('electron')
webContents.on('found-in-page', (event, result) => {
  if (result.finalUpdate) webContents.stopFindInPage('clearSelection')
})

const requestId = webContents.findInPage('api')
console.log(requestId)
```

`contents.capturePage([rect, ]callback)`

- `rect` Rectangle (optional) - The area of the page to be captured
- `callback` Function
    - `image` NativeImage

Captures a snapshot of the page within `rect` . Upon completion `callback` will be called with `callback(image)` . The `image` is an instance of NativeImage that stores data of the snapshot. Omitting `rect` will capture the whole visible page.

`contents.hasServiceWorker(callback)`

- `callback` Function
    - `hasWorker` Boolean

Checks if any ServiceWorker is registered and returns a boolean as response to `callback` .

`contents.unregisterServiceWorker(callback)`

- `callback` Function
    - `success` Boolean

Unregisters any ServiceWorker if present and returns a boolean as response to `callback` when the JS promise is fulfilled or false when the JS promise is rejected.

`contents.print([options])`

- `options` Object (optional)
    - `silent` Boolean - Don't ask user for print settings. Default is `false` .
    - `printBackground` Boolean - Also prints the background color and image of the web page. Default is `false` .

Prints window's web page. When `silent` is set to `true` , Electron will pick up system's default printer and default settings for printing.

Calling `window.print()` in web page is equivalent to calling `webContents.print({silent: false, printBackground: false})` .

Use `page-break-before: always;` CSS style to force to print to a new page.

`contents.printToPDF(options, callback)`

- `options` Object
  - `marginsType` Integer - (optional) Specifies the type of margins to use. Uses 0 for default margin, 1 for no margin, and 2 for minimum margin.
  - `pageSize` String - (optional) Specify page size of the generated PDF. Can be `A3` , `A4` , `A5` , `Legal` , `Letter` , `Tabloid` or an Object containing `height` and `width` in microns.
  - `printBackground` Boolean - (optional) Whether to print CSS backgrounds.
  - `printSelectionOnly` Boolean - (optional) Whether to print selection only.
  - `landscape` Boolean - (optional) `true` for landscape, `false` for portrait.
- `callback` Function
  - `error` Error
  - `data` Buffer

Prints window's web page as PDF with Chromium's preview printing custom settings.

The `callback` will be called with `callback(error, data)` on completion. The `data` is a `Buffer` that contains the generated PDF data.

The `landscape` will be ignored if `@page` CSS at-rule is used in the web page.

By default, an empty `options` will be regarded as:

```
{
  marginsType: 0,
  printBackground: false,
  printSelectionOnly: false,
  landscape: false
}
```

Use `page-break-before: always;` CSS style to force to print to a new page.

An example of `webContents.printToPDF` :

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

let win = new BrowserWindow({width: 800, height: 600})
win.loadURL('http://github.com')

win.webContents.on('did-finish-load', () => {
  // Use default printing options
  win.webContents.printToPDF({}, (error, data) => {
    if (error) throw error
    fs.writeFile('/tmp/print.pdf', data, (error) => {
      if (error) throw error
      console.log('Write PDF successfully.')
    })
  })
})
```

`contents.addWorkSpace(path)`

- `path` String

Adds the specified path to DevTools workspace. Must be used after DevTools creation:

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()
win.webContents.on('devtools-opened', () => {
  win.webContents.addWorkSpace(__dirname)
})
```

`contents.removeWorkSpace(path)`

- `path` String

Removes the specified path from DevTools workspace.

`contents.openDevTools([options])`

- `options` Object (optional)
  - `mode` String - Opens the devtools with specified dock state, can be `right`, `bottom`, `undocked`, `detach`. Defaults to last used dock state. In `undocked` mode it's possible to dock back. In `detach` mode it's not.

Opens the devtools.

`contents.closeDevTools()`

Closes the devtools.

`contents.isDevToolsOpened()`

Returns `Boolean` - Whether the devtools is opened.

`contents.isDevToolsFocused()`

Returns `Boolean` - Whether the devtools view is focused .

`contents.toggleDevTools()`

Toggles the developer tools.

`contents.inspectElement(x, y)`

- `x` Integer
- `y` Integer

Starts inspecting element at position (`x` , `y`).

`contents.inspectServiceWorker()`

Opens the developer tools for the service worker context.

`contents.send(channel[, arg1][, arg2][, ...])`

- `channel` String
- `...args` any[]

Send an asynchronous message to renderer process via `channel` , you can also send arbitrary arguments. Arguments will be serialized in JSON internally and hence no functions or prototype chain will be included.

The renderer process can handle the message by listening to `channel` with the `ipcRenderer` module.

An example of sending messages from the main process to the renderer process:

```javascript
// In the main process.
const {app, BrowserWindow} = require('electron')
let win = null

app.on('ready', () => {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(`file://${__dirname}/index.html`)
  win.webContents.on('did-finish-load', () => {
    win.webContents.send('ping', 'whoooooooh!')
  })
})
```

```html
<!-- index.html -->
<html>
<body>
  <script>
    require('electron').ipcRenderer.on('ping', (event, message) => {
      console.log(message)  // Prints 'whoooooooh!'
    })
  </script>
</body>
</html>
```

`contents.enableDeviceEmulation(parameters)`

- `parameters` Object
  - `screenPosition` String - Specify the screen type to emulate (default: `desktop`)
    - `desktop` - Desktop screen type
    - `mobile` - Mobile screen type
  - `screenSize` Object - Set the emulated screen size (screenPosition == mobile)
    - `width` Integer - Set the emulated screen width
    - `height` Integer - Set the emulated screen height
  - `viewPosition` Object - Position the view on the screen (screenPosition == mobile) (default: `{x: 0, y: 0}`)
    - `x` Integer - Set the x axis offset from top left corner
    - `y` Integer - Set the y axis offset from top left corner
  - `deviceScaleFactor` Integer - Set the device scale factor (if zero defaults to original device scale factor) (default: `0`)
  - `viewSize` Object - Set the emulated view size (empty means no override)
    - `width` Integer - Set the emulated view width
    - `height` Integer - Set the emulated view height
  - `fitToView` Boolean - Whether emulated view should be scaled down if necessary to fit into available space (default: `false`)
  - `offset` Object - Offset of the emulated view inside available space (not in fit to view mode) (default: `{x: 0, y: 0}`)
    - `x` Float - Set the x axis offset from top left corner
    - `y` Float - Set the y axis offset from top left corner
  - `scale` Float - Scale of emulated view inside available space (not in fit to view mode) (default: `1`)

Enable device emulation with the given parameters.

`contents.disableDeviceEmulation()`

Disable device emulation enabled by `webContents.enableDeviceEmulation` .

`contents.sendInputEvent(event)`

- `event` Object
  - `type` String (**required**) - The type of the event, can be `mouseDown` , `mouseUp` , `mouseEnter` , `mouseLeave` ,

`contextMenu` , `mouseWheel` , `mouseMove` , `keyDown` , `keyUp` , `char` .
  - `modifiers` String[] - An array of modifiers of the event, can include `shift` , `control` , `alt` , `meta` , `isKeypad` , `isAutoRepeat` , `leftButtonDown` , `middleButtonDown` , `rightButtonDown` , `capsLock` , `numLock` , `left` , `right` .

Sends an input `event` to the page.

For keyboard events, the `event` object also have following properties:

- `keyCode` String (**required**) - The character that will be sent as the keyboard event. Should only use the valid key codes in Accelerator.

For mouse events, the `event` object also have following properties:

- `x` Integer (**required**)
- `y` Integer (**required**)
- `button` String - The button pressed, can be `left` , `middle` , `right`
- `globalX` Integer
- `globalY` Integer
- `movementX` Integer
- `movementY` Integer
- `clickCount` Integer

For the `mouseWheel` event, the `event` object also have following properties:

- `deltaX` Integer
- `deltaY` Integer
- `wheelTicksX` Integer
- `wheelTicksY` Integer
- `accelerationRatioX` Integer
- `accelerationRatioY` Integer
- `hasPreciseScrollingDeltas` Boolean
- `canScroll` Boolean

`contents.beginFrameSubscription([onlyDirty ,]callback)`

- `onlyDirty` Boolean (optional) - Defaults to `false`
- `callback` Function
  - `frameBuffer` Buffer
  - `dirtyRect` Rectangle

Begin subscribing for presentation events and captured frames, the `callback` will be called with `callback(frameBuffer, dirtyRect)` when there is a presentation event.

The `frameBuffer` is a `Buffer` that contains raw pixel data. On most machines, the pixel data is effectively stored in 32bit BGRA format, but the actual representation depends on the endianness of the processor (most modern processors are little-endian, on machines with big-endian processors the data is in 32bit ARGB format).

The `dirtyRect` is an object with `x, y, width, height` properties that describes which part of the page was repainted. If `onlyDirty` is set to `true` , `frameBuffer` will only contain the repainted area. `onlyDirty` defaults to `false` .

`contents.endFrameSubscription()`

End subscribing for frame presentation events.

`contents.startDrag(item)`

- `item` Object
  - `file` String or `files` Array - The path(s) to the file(s) being dragged.
  - `icon` NativeImage - The image must be non-empty on macOS.

Sets the `item` as dragging item for current drag-drop operation, `file` is the absolute path of the file to be dragged, and `icon` is the image showing under the cursor when dragging.

`contents.savePage(fullPath, saveType, callback)`

- `fullPath` String - The full file path.
- `saveType` String - Specify the save type.
  - `HTMLOnly` - Save only the HTML of the page.
  - `HTMLComplete` - Save complete-html page.
  - `MHTML` - Save complete-html page as MHTML.
- `callback` Function - `(error) => {}`.
  - `error` Error

Returns `Boolean` - true if the process of saving page has been initiated successfully.

```
const {BrowserWindow} = require('electron')
let win = new BrowserWindow()

win.loadURL('https://github.com')

win.webContents.on('did-finish-load', () => {
  win.webContents.savePage('/tmp/test.html', 'HTMLComplete', (error) => {
    if (!error) console.log('Save page successfully')
  })
})
```

`contents.showDefinitionForSelection()` *macOS*

Shows pop-up dictionary that searches the selected word on the page.

`contents.setSize(options)`

Set the size of the page. This is only supported for `<webview>` guest contents.

- `options` Object
  - `normal` Object (optional) - Normal size of the page. This can be used in combination with the `disableguestresize` attribute to manually resize the webview guest contents.
    - `width` Integer
    - `height` Integer

`contents.isOffscreen()`

Returns `Boolean` - Indicates whether *offscreen rendering* is enabled.

`contents.startPainting()`

If *offscreen rendering* is enabled and not painting, start painting.

`contents.stopPainting()`

If *offscreen rendering* is enabled and painting, stop painting.

`contents.isPainting()`

Returns `Boolean` - If *offscreen rendering* is enabled returns whether it is currently painting.

`contents.setFrameRate(fps)`

- `fps` Integer

If *offscreen rendering* is enabled sets the frame rate to the specified number. Only values between 1 and 60 are accepted.

`contents.getFrameRate()`

Returns `Integer` - If *offscreen rendering* is enabled returns the current frame rate.

`contents.invalidate()`

Schedules a full repaint of the window this web contents is in.

If *offscreen rendering* is enabled invalidates the frame and generates a new one through the `'paint'` event.

Instance Properties

`contents.id`

A Integer representing the unique ID of this WebContents.

`contents.session`

A Session object (session) used by this webContents.

`contents.hostWebContents`

A `WebContents` instance that might own this `WebContents`.

`contents.devToolsWebContents`

A `WebContents` of DevTools for this `WebContents`.

Note: Users should never store this object because it may become `null` when the DevTools has been closed.

`contents.debugger`

A Debugger instance for this webContents.

desktopCapturer

> Access information about media sources that can be used to capture audio and video from the desktop using the `navigator.webkitGetUserMedia` API.

Process: Renderer

The following example shows how to capture video from a desktop window whose title is `Electron`:

```
// In the renderer process.
const {desktopCapturer} = require('electron')

desktopCapturer.getSources({types: ['window', 'screen']}, (error, sources) => {
  if (error) throw error
  for (let i = 0; i < sources.length; ++i) {
    if (sources[i].name === 'Electron') {
      navigator.webkitGetUserMedia({
        audio: false,
        video: {
          mandatory: {
            chromeMediaSource: 'desktop',
            chromeMediaSourceId: sources[i].id,
            minWidth: 1280,
            maxWidth: 1280,
            minHeight: 720,
            maxHeight: 720
          }
        }
      }, handleStream, handleError)
      return
    }
  }
})

function handleStream (stream) {
  document.querySelector('video').src = URL.createObjectURL(stream)
}

function handleError (e) {
  console.log(e)
}
```

To capture video from a source provided by `desktopCapturer` the constraints passed to `navigator.webkitGetUserMedia` must include `chromeMediaSource: 'desktop'`, and `audio: false`.

To capture both audio and video from the entire desktop the constraints passed to `navigator.webkitGetUserMedia` must include `chromeMediaSource: 'screen'`, and `audio: true`, but should not include a `chromeMediaSourceId` constraint.

Methods

The `desktopCapturer` module has the following methods:

`desktopCapturer.getSources(options, callback)`

- `options` Object
  - `types` String[] - An array of Strings that lists the types of desktop sources to be captured, available types are `screen` and `window`.
  - `thumbnailSize` Object (optional) - The suggested size that the media source thumbnail should be scaled to, defaults to `{width: 150, height: 150}`.
- `callback` Function

  - `error` Error
  - `sources` DesktopCapturerSource[]

Starts gathering information about all available desktop media sources, and calls `callback(error, sources)` when finished.

`sources` is an array of `DesktopCapturerSource` objects, each `DesktopCapturerSource` represents a screen or an individual window that can be captured.

ipcRenderer

> Communicate asynchronously from a renderer process to the main process.

Process: Renderer

The `ipcRenderer` module is an instance of the EventEmitter class. It provides a few methods so you can send synchronous and asynchronous messages from the render process (web page) to the main process. You can also receive replies from the main process.

See ipcMain for code examples.

Methods

The `ipcRenderer` module has the following method to listen for events and send messages:

`ipcRenderer.on(channel, listener)`

- `channel` String
- `listener` Function

Listens to `channel`, when a new message arrives `listener` would be called with `listener(event, args...)`.

`ipcRenderer.once(channel, listener)`

- `channel` String
- `listener` Function

Adds a one time `listener` function for the event. This `listener` is invoked only the next time a message is sent to `channel`, after which it is removed.

`ipcRenderer.removeListener(channel, listener)`

- `channel` String
- `listener` Function

Removes the specified `listener` from the listener array for the specified `channel`.

`ipcRenderer.removeAllListeners([channel])`

- `channel` String (optional)

Removes all listeners, or those of the specified `channel`.

`ipcRenderer.send(channel[, arg1][, arg2][, ...])`

- `channel` String
- `...args` any[]

Send a message to the main process asynchronously via `channel`, you can also send arbitrary arguments. Arguments will be serialized in JSON internally and hence no functions or prototype chain will be included.

The main process handles it by listening for `channel` with `ipcMain` module.

`ipcRenderer.sendSync(channel[, arg1][, arg2][, ...])`

* `channel` String
* `...args` any[]

Send a message to the main process synchronously via `channel`, you can also send arbitrary arguments. Arguments will be serialized in JSON internally and hence no functions or prototype chain will be included.

The main process handles it by listening for `channel` with `ipcMain` module, and replies by setting `event.returnValue`.

Note: Sending a synchronous message will block the whole renderer process, unless you know what you are doing you should never use it.

`ipcRenderer.sendToHost(channel[, arg1][, arg2][, ...])`

* `channel` String
* `...args` any[]

Like `ipcRenderer.send` but the event will be sent to the `<webview>` element in the host page instead of the main process.

remote

> Use main process modules from the renderer process.

Process: Renderer

The `remote` module provides a simple way to do inter-process communication (IPC) between the renderer process (web page) and the main process.

In Electron, GUI-related modules (such as `dialog`, `menu` etc.) are only available in the main process, not in the renderer process. In order to use them from the renderer process, the `ipc` module is necessary to send inter-process messages to the main process. With the `remote` module, you can invoke methods of the main process object without explicitly sending inter-process messages, similar to Java's RMI. An example of creating a browser window from a renderer process:

```
const {BrowserWindow} = require('electron').remote
let win = new BrowserWindow({width: 800, height: 600})
win.loadURL('https://github.com')
```

Note: For the reverse (access the renderer process from the main process), you can use webContents.executeJavascript.

Remote Objects

Each object (including functions) returned by the `remote` module represents an object in the main process (we call it a remote object or remote function). When you invoke methods of a remote object, call a remote function, or create a new object with the remote constructor (function), you are actually sending synchronous inter-process messages.

In the example above, both `BrowserWindow` and `win` were remote objects and `new BrowserWindow` didn't create a `BrowserWindow` object in the renderer process. Instead, it created a `BrowserWindow` object in the main process and returned the corresponding remote object in the renderer process, namely the `win` object.

Note: Only enumerable properties which are present when the remote object is first referenced are accessible via remote.

Note: Arrays and Buffers are copied over IPC when accessed via the `remote` module. Modifying them in the renderer process does not modify them in the main process and vice versa.

Lifetime of Remote Objects

Electron makes sure that as long as the remote object in the renderer process lives (in other words, has not been garbage collected), the corresponding object in the main process will not be released. When the remote object has been garbage collected, the corresponding object in the main process will be dereferenced.

If the remote object is leaked in the renderer process (e.g. stored in a map but never freed), the corresponding object in the main process will also be leaked, so you should be very careful not to leak remote objects.

Primary value types like strings and numbers, however, are sent by copy.

Passing callbacks to the main process

Code in the main process can accept callbacks from the renderer - for instance the `remote` module - but you should be extremely careful when using this feature.

First, in order to avoid deadlocks, the callbacks passed to the main process are called asynchronously. You should not expect the main process to get the return value of the passed callbacks.

For instance you can't use a function from the renderer process in an `Array.map` called in the main process:

```
// main process mapNumbers.js
exports.withRendererCallback = (mapper) => {
  return [1, 2, 3].map(mapper)
}

exports.withLocalCallback = () => {
  return [1, 2, 3].map(x => x + 1)
}
```

```
// renderer process
const mapNumbers = require('electron').remote.require('./mapNumbers')
const withRendererCb = mapNumbers.withRendererCallback(x => x + 1)
const withLocalCb = mapNumbers.withLocalCallback()

console.log(withRendererCb, withLocalCb)
// [undefined, undefined, undefined], [2, 3, 4]
```

As you can see, the renderer callback's synchronous return value was not as expected, and didn't match the return value of an identical callback that lives in the main process.

Second, the callbacks passed to the main process will persist until the main process garbage-collects them.

For example, the following code seems innocent at first glance. It installs a callback for the `close` event on a remote object:

```
require('electron').remote.getCurrentWindow().on('close', () => {
  // window was closed...
})
```

But remember the callback is referenced by the main process until you explicitly uninstall it. If you do not, each time you reload your window the callback will be installed again, leaking one callback for each restart.

To make things worse, since the context of previously installed callbacks has been released, exceptions will be raised in the main process when the `close` event is emitted.

To avoid this problem, ensure you clean up any references to renderer callbacks passed to the main process. This involves cleaning up event handlers, or ensuring the main process is explicitly told to deference callbacks that came from a renderer process that is exiting.

Accessing built-in modules in the main process

The built-in modules in the main process are added as getters in the `remote` module, so you can use them directly like the `electron` module.

```
const app = require('electron').remote.app
console.log(app)
```

Methods

The `remote` module has the following methods:

`remote.require(module)`

- `module` String

Returns `any` - The object returned by `require(module)` in the main process.

`remote.getCurrentWindow()`

Returns `BrowserWindow` - The window to which this web page belongs.

`remote.getCurrentWebContents()`

Returns `WebContents` - The web contents of this web page.

`remote.getGlobal(name)`

- `name` String

Returns `any` - The global variable of `name` (e.g. `global[name]`) in the main process.

Properties

`remote.process`

The `process` object in the main process. This is the same as `remote.getGlobal('process')` but is cached.

webFrame

> Customize the rendering of the current web page.

Process: Renderer

An example of zooming current page to 200%.

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

webFrame.setZoomFactor(2)
```

Methods

The `webFrame` module has the following methods:

`webFrame.setZoomFactor(factor)`

- `factor` Number - Zoom factor.

Changes the zoom factor to the specified factor. Zoom factor is zoom percent divided by 100, so 300% = 3.0.

`webFrame.getZoomFactor()`

Returns `Number` - The current zoom factor.

`webFrame.setZoomLevel(level)`

- `level` Number - Zoom level

Changes the zoom level to the specified level. The original size is 0 and each increment above or below represents zooming 20% larger or smaller to default limits of 300% and 50% of original size, respectively.

`webFrame.getZoomLevel()`

Returns `Number` - The current zoom level.

`webFrame.setZoomLevelLimits(minimumLevel, maximumLevel)`

- `minimumLevel` Number
- `maximumLevel` Number

Deprecated: Call `setVisualZoomLevelLimits` instead to set the visual zoom level limits. This method will be removed in Electron 2.0.

`webFrame.setVisualZoomLevelLimits(minimumLevel, maximumLevel)`

- `minimumLevel` Number
- `maximumLevel` Number

Sets the maximum and minimum pinch-to-zoom level.

`webFrame.setLayoutZoomLevelLimits(minimumLevel, maximumLevel)`

- `minimumLevel` Number

- `maximumLevel` Number

Sets the maximum and minimum layout-based (i.e. non-visual) zoom level.

`webFrame.setSpellCheckProvider(language, autoCorrectWord, provider)`

- `language` String
- `autoCorrectWord` Boolean
- `provider` Object
  - `spellCheck` Function - Returns `Boolean`
    - `text` String

Sets a provider for spell checking in input fields and text areas.

The `provider` must be an object that has a `spellCheck` method that returns whether the word passed is correctly spelled.

An example of using node-spellchecker as provider:

```
const {webFrame} = require('electron')
webFrame.setSpellCheckProvider('en-US', true, {
  spellCheck (text) {
    return !(require('spellchecker').isMisspelled(text))
  }
})
```

`webFrame.registerURLSchemeAsSecure(scheme)`

- `scheme` String

Registers the `scheme` as secure scheme.

Secure schemes do not trigger mixed content warnings. For example, `https` and `data` are secure schemes because they cannot be corrupted by active network attackers.

`webFrame.registerURLSchemeAsBypassingCSP(scheme)`

- `scheme` String

Resources will be loaded from this `scheme` regardless of the current page's Content Security Policy.

`webFrame.registerURLSchemeAsPrivileged(scheme[, options])`

- `scheme` String
- `options` Object (optional)
  - `secure` Boolean - (optional) Default true.
  - `bypassCSP` Boolean - (optional) Default true.
  - `allowServiceWorkers` Boolean - (optional) Default true.
  - `supportFetchAPI` Boolean - (optional) Default true.
  - `corsEnabled` Boolean - (optional) Default true.

Registers the `scheme` as secure, bypasses content security policy for resources, allows registering ServiceWorker and supports fetch API.

Specify an option with the value of `false` to omit it from the registration. An example of registering a privileged scheme, without bypassing Content Security Policy:

```
const {webFrame} = require('electron')
webFrame.registerURLSchemeAsPrivileged('foo', { bypassCSP: false })
```

`webFrame.insertText(text)`

- `text` String

Inserts `text` to the focused element.

`webFrame.executeJavaScript(code[, userGesture, callback])`

- `code` String
- `userGesture` Boolean (optional) - Default is `false` .
- `callback` Function (optional) - Called after script has been executed.
    - `result` Any

Evaluates `code` in page.

In the browser window some HTML APIs like `requestFullScreen` can only be invoked by a gesture from the user. Setting `userGesture` to `true` will remove this limitation.

`webFrame.getResourceUsage()`

Returns `Object` :

- `images` MemoryUsageDetails
- `cssStyleSheets` MemoryUsageDetails
- `xslStyleSheets` MemoryUsageDetails
- `fonts` MemoryUsageDetails
- `other` MemoryUsageDetails

Returns an object describing usage information of Blink's internal memory caches.

```
const {webFrame} = require('electron')
console.log(webFrame.getResourceUsage())
```

This will generate:

```
{
  images: {
    count: 22,
    size: 2549,
    liveSize: 2542
  },
  cssStyleSheets: { /* same with "images" */ },
  xslStyleSheets: { /* same with "images" */ },
  fonts: { /* same with "images" */ },
  other: { /* same with "images" */ }
}
```

`webFrame.clearCache()`

Attempts to free memory that is no longer being used (like images from a previous navigation).

Note that blindly calling this method probably makes Electron slower since it will have to refill these emptied caches, you should only call it if an event in your app has occurred that makes you think your page is actually using less memory (i.e. you have navigated from a super heavy page to a mostly empty one, and intend to stay there).

clipboard

> Perform copy and paste operations on the system clipboard.

Process: Main, Renderer

The following example shows how to write a string to the clipboard:

```
const {clipboard} = require('electron')
clipboard.writeText('Example String')
```

On X Window systems, there is also a selection clipboard. To manipulate it you need to pass `selection` to each method:

```
const {clipboard} = require('electron')
clipboard.writeText('Example String', 'selection')
console.log(clipboard.readText('selection'))
```

Methods

The `clipboard` module has the following methods:

Note: Experimental APIs are marked as such and could be removed in future.

`clipboard.readText([type])`

- `type` String (optional)

Returns `String` - The content in the clipboard as plain text.

`clipboard.writeText(text[, type])`

- `text` String
- `type` String (optional)

Writes the `text` into the clipboard as plain text.

`clipboard.readHTML([type])`

- `type` String (optional)

Returns `String` - The content in the clipboard as markup.

`clipboard.writeHTML(markup[, type])`

- `markup` String
- `type` String (optional)

Writes `markup` to the clipboard.

`clipboard.readImage([type])`

- `type` String (optional)

Returns `NativeImage` - The image content in the clipboard.

`clipboard.writeImage(image[, type])`

- `image` NativeImage
- `type` String (optional)

Writes `image` to the clipboard.

`clipboard.readRTF([type])`

- `type` String (optional)

Returns `String` - The content in the clipboard as RTF.

`clipboard.writeRTF(text[, type])`

- `text` String
- `type` String (optional)

Writes the `text` into the clipboard in RTF.

`clipboard.readBookmark()` *macOS Windows*

Returns `Object` :

- `title` String
- `url` String

Returns an Object containing `title` and `url` keys representing the bookmark in the clipboard. The `title` and `url` values will be empty strings when the bookmark is unavailable.

`clipboard.writeBookmark(title, url[, type])` *macOS Windows*

- `title` String
- `url` String
- `type` String (optional)

Writes the `title` and `url` into the clipboard as a bookmark.

Note: Most apps on Windows don't support pasting bookmarks into them so you can use `clipboard.write` to write both a bookmark and fallback text to the clipboard.

```
clipboard.write({
  text: 'https://electron.atom.io',
  bookmark: 'Electron Homepage'
})
```

`clipboard.readFindText()` *macOS*

Returns `String` - The text on the find pasteboard. This method uses synchronous IPC when called from the renderer process. The cached value is reread from the find pasteboard whenever the application is activated.

`clipboard.writeFindText(text)` *macOS*

- `text` String

Writes the `text` into the find pasteboard as plain text. This method uses synchronous IPC when called from the renderer process.

`clipboard.clear([type])`

- `type` String (optional)

Clears the clipboard content.

`clipboard.availableFormats([type])`

- `type` String (optional)

Returns `String[]` - An array of supported formats for the clipboard `type` .

`clipboard.has(data[, type])` *Experimental*

- `data` String
- `type` String (optional)

Returns `Boolean` - Whether the clipboard supports the format of specified `data` .

```
const {clipboard} = require('electron')
console.log(clipboard.has('<p>selection</p>'))
```

`clipboard.read(data[, type])` *Experimental*

- `data` String
- `type` String (optional)

Returns `String` - Reads `data` from the clipboard.

`clipboard.write(data[, type])`

- `data` Object
  - `text` String (optional)
  - `html` String (optional)
  - `image` NativeImage (optional)
  - `rtf` String (optional)
  - `bookmark` String (optional) - The title of the url at `text` .
- `type` String (optional)

```
const {clipboard} = require('electron')
clipboard.write({text: 'test', html: '<b>test</b>'})
```

Writes `data` to the clipboard.

crashReporter

> Submit crash reports to a remote server.

Process: Main, Renderer

The following is an example of automatically submitting a crash report to a remote server:

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

crashReporter.start({
  productName: 'YourName',
  companyName: 'YourCompany',
  submitURL: 'https://your-domain.com/url-to-submit',
  uploadToServer: true
})
```

For setting up a server to accept and process crash reports, you can use following projects:

- socorro
- mini-breakpad-server

Crash reports are saved locally in an application-specific temp directory folder. For a `productName` of `YourName`, crash reports will be stored in a folder named `YourName Crashes` inside the temp directory. You can customize this temp directory location for your app by calling the `app.setPath('temp', '/my/custom/temp')` API before starting the crash reporter.

Methods

The `crashReporter` module has the following methods:

`crashReporter.start(options)`

- `options` Object
  - `companyName` String (optional)
  - `submitURL` String - URL that crash reports will be sent to as POST.
  - `productName` String (optional) - Defaults to `app.getName()`.
  - `uploadToServer` Boolean (optional) *macOS* - Whether crash reports should be sent to the server Default is `true`.
  - `ignoreSystemCrashHandler` Boolean (optional) - Default is `false`.
  - `extra` Object (optional) - An object you can define that will be sent along with the report. Only string properties are sent correctly. Nested objects are not supported.

You are required to call this method before using any other `crashReporter` APIs and in each process (main/renderer) from which you want to collect crash reports. You can pass different options to `crashReporter.start` when calling from different processes.

Note Child processes created via the `child_process` module will not have access to the Electron modules. Therefore, to collect crash reports from them, use `process.crashReporter.start` instead. Pass the same options as above along with an additional one called `crashesDirectory` that should point to a directory to store the crash reports temporarily. You can test this out by calling `process.crash()` to crash the child process.

Note: To collect crash reports from child process in Windows, you need to add this extra code as well. This will start the process that will monitor and send the crash reports. Replace `submitURL`, `productName` and `crashesDirectory` with appropriate values.

Note: If you need send additional/updated `extra` parameters after your first call `start` you can call `setExtraParameter` on macOS or call `start` again with the new/updated `extra` parameters on Linux and Windows.

```
const args = [
  `--reporter-url=${submitURL}`,
  `--application-name=${productName}`,
  `--crashes-directory=${crashesDirectory}`
]
const env = {
  ELECTRON_INTERNAL_CRASH_SERVICE: 1
}
spawn(process.execPath, args, {
  env: env,
  detached: true
})
```

Note: On macOS, Electron uses a new `crashpad` client for crash collection and reporting. If you want to enable crash reporting, initializing `crashpad` from the main process using `crashReporter.start` is required regardless of which process you want to collect crashes from. Once initialized this way, the crashpad handler collects crashes from all processes. You still have to call `crashReporter.start` from the renderer or child process, otherwise crashes from them will get reported without `companyName` , `productName` or any of the `extra` information.

`crashReporter.getLastCrashReport()`

Returns `CrashReport` :

Returns the date and ID of the last crash report. If no crash reports have been sent or the crash reporter has not been started, `null` is returned.

`crashReporter.getUploadedReports()`

Returns `CrashReport[]` :

Returns all uploaded crash reports. Each report contains the date and uploaded ID.

`crashReporter.getUploadToServer()` *macOS*

Returns `Boolean` - Whether reports should be submitted to the server. Set through the `start` method or `setUploadToServer` .

Note: This API can only be called from the main process.

`crashReporter.setUploadToServer(uploadToServer)` *macOS*

- `uploadToServer` Boolean *macOS* - Whether reports should be submitted to the server

This would normally be controlled by user preferences. This has no effect if called before `start` is called.

Note: This API can only be called from the main process.

`crashReporter.setExtraParameter(key, value)` *macOS*

- `key` String - Parameter key.
- `value` String - Parameter value. Specifying `null` or `undefined` will remove the key from the extra parameters.

Set an extra parameter to set be sent with the crash report. The values specified here will be sent in addition to any values set via the `extra` option when `start` was called. This API is only available on macOS, if you need to add/update extra parameters on Linux and Windows after your first call to `start` you can call `start` again with the updated `extra` options.

Crash Report Payload

The crash reporter will send the following data to the `submitURL` as a `multipart/form-data` `POST` :

- `ver` String - The version of Electron.
- `platform` String - e.g. 'win32'.
- `process_type` String - e.g. 'renderer'.
- `guid` String - e.g. '5e1286fc-da97-479e-918b-6bfb0c3d1c72'
- `_version` String - The version in `package.json` .
- `_productName` String - The product name in the `crashReporter` `options` object.
- `prod` String - Name of the underlying product. In this case Electron.
- `_companyName` String - The company name in the `crashReporter` `options` object.
- `upload_file_minidump` File - The crash report in the format of `minidump` .
- All level one properties of the `extra` object in the `crashReporter` `options` object.

nativeImage

> Create tray, dock, and application icons using PNG or JPG files.

Process: Main, Renderer

In Electron, for the APIs that take images, you can pass either file paths or `NativeImage` instances. An empty image will be used when `null` is passed.

For example, when creating a tray or setting a window's icon, you can pass an image file path as a `String` :

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

const appIcon = new Tray('/Users/somebody/images/icon.png')
let win = new BrowserWindow({icon: '/Users/somebody/images/window.png'})
console.log(appIcon, win)
```

Or read the image from the clipboard which returns a `NativeImage` :

```
const {clipboard, Tray} = require('electron')
const image = clipboard.readImage()
const appIcon = new Tray(image)
console.log(appIcon)
```

Supported Formats

Currently `PNG` and `JPEG` image formats are supported. `PNG` is recommended because of its support for transparency and lossless compression.

On Windows, you can also load `ICO` icons from file paths. For best visual quality it is recommended to include at least the following sizes in the:

- Small icon
  - 16x16 (100% DPI scale)
  - 20x20 (125% DPI scale)
  - 24x24 (150% DPI scale)
  - 32x32 (200% DPI scale)
- Large icon
  - 32x32 (100% DPI scale)
  - 40x40 (125% DPI scale)
  - 48x48 (150% DPI scale)
  - 64x64 (200% DPI scale)
- 256x256

Check the *Size requirements* section in this article.

High Resolution Image

On platforms that have high-DPI support such as Apple Retina displays, you can append `@2x` after image's base filename to mark it as a high resolution image.

For example if `icon.png` is a normal image that has standard resolution, then `icon@2x.png` will be treated as a high resolution image that has double DPI density.

If you want to support displays with different DPI densities at the same time, you can put images with different sizes in the same folder and use the filename without DPI suffixes. For example:

```
images/
├── icon.png
├── icon@2x.png
└── icon@3x.png
```

```
const {Tray} = require('electron')
let appIcon = new Tray('/Users/somebody/images/icon.png')
console.log(appIcon)
```

Following suffixes for DPI are also supported:

- `@1x`
- `@1.25x`
- `@1.33x`
- `@1.4x`
- `@1.5x`
- `@1.8x`
- `@2x`
- `@2.5x`
- `@3x`
- `@4x`
- `@5x`

Template Image

Template images consist of black and clear colors (and an alpha channel). Template images are not intended to be used as standalone images and are usually mixed with other content to create the desired final appearance.

The most common case is to use template images for a menu bar icon so it can adapt to both light and dark menu bars.

Note: Template image is only supported on macOS.

To mark an image as a template image, its filename should end with the word `Template`. For example:

- `xxxTemplate.png`
- `xxxTemplate@2x.png`

Methods

The `nativeImage` module has the following methods, all of which return an instance of the `NativeImage` class:

`nativeImage.createEmpty()`

Returns `NativeImage`

Creates an empty `NativeImage` instance.

`nativeImage.createFromPath(path)`

- `path` String

Returns `NativeImage`

Creates a new `NativeImage` instance from a file located at `path`. This method returns an empty image if the `path` does not exist, cannot be read, or is not a valid image.

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

let image = nativeImage.createFromPath('/Users/somebody/images/icon.png')
console.log(image)
```

`nativeImage.createFromBuffer(buffer[, options])`

- `buffer` Buffer
- `options` Object (optional)
    - `width` Integer (optional) - Required for bitmap buffers.
    - `height` Integer (optional) - Required for bitmap buffers.
    - `scaleFactor` Double (optional) - Defaults to 1.0.

Returns `NativeImage`

Creates a new `NativeImage` instance from `buffer`.

`nativeImage.createFromDataURL(dataURL)`

- `dataURL` String

Creates a new `NativeImage` instance from `dataURL`.

Class: NativeImage

> Natively wrap images such as tray, dock, and application icons.

Process: Main, Renderer

Instance Methods

The following methods are available on instances of the `NativeImage` class:

`image.toPNG()`

Returns `Buffer` - A Buffer that contains the image's `PNG` encoded data.

`image.toJPEG(quality)`

- `quality` Integer (**required**) - Between 0 - 100.

Returns `Buffer` - A Buffer that contains the image's `JPEG` encoded data.

`image.toBitmap()`

Returns `Buffer` - A Buffer that contains a copy of the image's raw bitmap pixel data.

`image.toDataURL()`

Returns `String` - The data URL of the image.

`image.getBitmap()`

Returns `Buffer` - A Buffer that contains the image's raw bitmap pixel data.

The difference between `getBitmap()` and `toBitmap()` is, `getBitmap()` does not copy the bitmap data, so you have to use the returned Buffer immediately in current event loop tick, otherwise the data might be changed or destroyed.

`image.getNativeHandle()` *macOS*

Returns `Buffer` - A Buffer that stores C pointer to underlying native handle of the image. On macOS, a pointer to `NSImage` instance would be returned.

Notice that the returned pointer is a weak pointer to the underlying native image instead of a copy, so you *must* ensure that the associated `nativeImage` instance is kept around.

`image.isEmpty()`

Returns `Boolean` - Whether the image is empty.

`image.getSize()`

Returns `Object` :

- `width` Integer
- `height` Integer

`image.setTemplateImage(option)`

- `option` Boolean

Marks the image as a template image.

`image.isTemplateImage()`

Returns `Boolean` - Whether the image is a template image.

`image.crop(rect)`

- `rect` Object - The area of the image to crop
  - `x` Integer
  - `y` Integer
  - `width` Integer
  - `height` Integer

Returns `NativeImage` - The cropped image.

`image.resize(options)`

- `options` Object
  - `width` Integer (optional)
  - `height` Integer (optional)
  - `quality` String (optional) - The desired quality of the resize image. Possible values are `good` , `better` or `best` . The default is `best` . These values express a desired quality/speed tradeoff. They are translated into an algorithm-specific method that depends on the capabilities (CPU, GPU) of the underlying platform. It is possible for all three methods to be mapped to the same algorithm on a given platform.

Returns `NativeImage` - The resized image.

If only the `height` or the `width` are specified then the current aspect ratio will be preserved in the resized image.

`image.getAspectRatio()`

Returns `Float` - The image's aspect ratio.

screen

> Retrieve information about screen size, displays, cursor position, etc.

Process: Main, Renderer

You cannot require or use this module until the `ready` event of the `app` module is emitted.

`screen` is an EventEmitter.

Note: In the renderer / DevTools, `window.screen` is a reserved DOM property, so writing `let {screen} = require('electron')` will not work.

An example of creating a window that fills the whole screen:

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

let win

app.on('ready', () => {
  const {width, height} = electron.screen.getPrimaryDisplay().workAreaSize
  win = new BrowserWindow({width, height})
  win.loadURL('https://github.com')
})
```

Another example of creating a window in the external display:

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

let win

app.on('ready', () => {
  let displays = electron.screen.getAllDisplays()
  let externalDisplay = displays.find((display) => {
    return display.bounds.x !== 0 || display.bounds.y !== 0
  })

  if (externalDisplay) {
    win = new BrowserWindow({
      x: externalDisplay.bounds.x + 50,
      y: externalDisplay.bounds.y + 50
    })
    win.loadURL('https://github.com')
  }
})
```

Events

The `screen` module emits the following events:

Event: 'display-added'

Returns:

- `event` Event
- `newDisplay` Display

Emitted when `newDisplay` has been added.

Event: 'display-removed'

Returns:

- `event` Event
- `oldDisplay` Display

Emitted when `oldDisplay` has been removed.

Event: 'display-metrics-changed'

Returns:

- `event` Event
- `display` Display
- `changedMetrics` String[]

Emitted when one or more metrics change in a `display`. The `changedMetrics` is an array of strings that describe the changes. Possible changes are `bounds`, `workArea`, `scaleFactor` and `rotation`.

Methods

The `screen` module has the following methods:

`screen.getCursorScreenPoint()`

Returns `Object`:

- `x` Integer
- `y` Integer

The current absolute position of the mouse pointer.

`screen.getPrimaryDisplay()`

Returns `Display` - The primary display.

`screen.getAllDisplays()`

Returns `Display[]` - An array of displays that are currently available.

`screen.getDisplayNearestPoint(point)`

- `point` Object
  - `x` Integer
  - `y` Integer

Returns `Display` - The display nearest the specified point.

`screen.getDisplayMatching(rect)`

- `rect` Rectangle

Returns `Display` - The display that most closely intersects the provided bounds.

shell

> Manage files and URLs using their default applications.

Process: Main, Renderer

The `shell` module provides functions related to desktop integration.

An example of opening a URL in the user's default browser:

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

shell.openExternal('https://github.com')
```

Methods

The `shell` module has the following methods:

`shell.showItemInFolder(fullPath)`

- `fullPath` String

Returns `Boolean` - Whether the item was successfully shown

Show the given file in a file manager. If possible, select the file.

`shell.openItem(fullPath)`

- `fullPath` String

Returns `Boolean` - Whether the item was successfully opened.

Open the given file in the desktop's default manner.

`shell.openExternal(url[, options, callback])`

- `url` String
- `options` Object (optional) *macOS*
  - `activate` Boolean - `true` to bring the opened application to the foreground. The default is `true` .
- `callback` Function (optional) - If specified will perform the open asynchronously. *macOS*
  - `error` Error

Returns `Boolean` - Whether an application was available to open the URL. If callback is specified, always returns true.

Open the given external protocol URL in the desktop's default manner. (For example, mailto: URLs in the user's default mail agent).

`shell.moveItemToTrash(fullPath)`

- `fullPath` String

Returns `Boolean` - Whether the item was successfully moved to the trash

Move the given file to trash and returns a boolean status for the operation.

`shell.beep()`

Play the beep sound.

`shell.writeShortcutLink(shortcutPath[, operation], options)` *Windows*

- `shortcutPath` String
- `operation` String (optional) - Default is `create`, can be one of following:
  - `create` - Creates a new shortcut, overwriting if necessary.
  - `update` - Updates specified properties only on an existing shortcut.
  - `replace` - Overwrites an existing shortcut, fails if the shortcut doesn't exist.
- `options` ShortcutDetails

Returns `Boolean` - Whether the shortcut was created successfully

Creates or updates a shortcut link at `shortcutPath`.

`shell.readShortcutLink(shortcutPath)` *Windows*

- `shortcutPath` String

Returns `ShortcutDetails`

Resolves the shortcut link at `shortcutPath`.

An exception will be thrown when any error happens.

Coding Style

These are the style guidelines for coding in Electron.

You can run `npm run lint` to show any style issues detected by `cpplint` and `eslint`.

C++ and Python

For C++ and Python, we follow Chromium's Coding Style. You can use clang-format to format the C++ code automatically. There is also a script `script/cpplint.py` to check whether all files conform.

The Python version we are using now is Python 2.7.

The C++ code uses a lot of Chromium's abstractions and types, so it's recommended to get acquainted with them. A good place to start is Chromium's Important Abstractions and Data Structures document. The document mentions some special types, scoped types (that automatically release their memory when going out of scope), logging mechanisms etc.

JavaScript

- Write standard JavaScript style.
- File names should be concatenated with `-` instead of `_`, e.g. `file-name.js` rather than `file_name.js`, because in github/atom module names are usually in the `module-name` form. This rule only applies to `.js` files.
- Use newer ES6/ES2015 syntax where appropriate
  - `const` for requires and other constants
  - `let` for defining variables
  - Arrow functions instead of `function () { }`
  - Template literals instead of string concatenation using `+`

Naming Things

Electron APIs uses the same capitalization scheme as Node.js:

- When the module itself is a class like `BrowserWindow`, use `CamelCase`.
- When the module is a set of APIs, like `globalShortcut`, use `mixedCase`.
- When the API is a property of object, and it is complex enough to be in a separate chapter like `win.webContents`, use `mixedCase`.
- For other non-module APIs, use natural titles, like `<webview> Tag` or `Process Object`.

When creating a new API, it is preferred to use getters and setters instead of jQuery's one-function style. For example, `.getText()` and `.setText(text)` are preferred to `.text([text])`. There is a discussion on this.

Source Code Directory Structure

The source code of Electron is separated into a few parts, mostly following Chromium on the separation conventions.

You may need to become familiar with Chromium's multi-process architecture to understand the source code better.

Structure of Source Code

```
Electron
├── atom/ - C++ source code.
|   ├── app/ - System entry code.
|   ├── browser/ - The frontend including the main window, UI, and all of the
|   |   main process things. This talks to the renderer to manage web pages.
|   |   ├── ui/ - Implementation of UI stuff for different platforms.
|   |   |   ├── cocoa/ - Cocoa specific source code.
|   |   |   ├── win/ - Windows GUI specific source code.
|   |   |   └── x/ - X11 specific source code.
|   |   ├── api/ - The implementation of the main process APIs.
|   |   ├── net/ - Network related code.
|   |   ├── mac/ - Mac specific Objective-C source code.
|   |   └── resources/ - Icons, platform-dependent files, etc.
|   ├── renderer/ - Code that runs in renderer process.
|   |   └── api/ - The implementation of renderer process APIs.
|   └── common/ - Code that used by both the main and renderer processes,
|       including some utility functions and code to integrate node's message
|       loop into Chromium's message loop.
|       └── api/ - The implementation of common APIs, and foundations of
|           Electron's built-in modules.
├── chromium_src/ - Source code that copied from Chromium.
├── default_app/ - The default page to show when Electron is started without
|   providing an app.
├── docs/ - Documentations.
├── lib/ - JavaScript source code.
|   ├── browser/ - Javascript main process initialization code.
|   |   └── api/ - Javascript API implementation.
|   ├── common/ - JavaScript used by both the main and renderer processes
|   |   └── api/ - Javascript API implementation.
|   └── renderer/ - Javascript renderer process initialization code.
|       └── api/ - Javascript API implementation.
├── spec/ - Automatic tests.
├── electron.gyp - Building rules of Electron.
└── common.gypi - Compiler specific settings and building rules for other
    components like `node` and `breakpad`.
```

Structure of Other Directories

- **script** - Scripts used for development purpose like building, packaging, testing, etc.
- **tools** - Helper scripts used by gyp files, unlike `script`, scripts put here should never be invoked by users directly.
- **vendor** - Source code of third party dependencies, we didn't use `third_party` as name because it would confuse it with the same directory in Chromium's source code tree.
- **node_modules** - Third party node modules used for building.
- **out** - Temporary output directory of `ninja`.
- **dist** - Temporary directory created by `script/create-dist.py` script when creating a distribution.
- **external_binaries** - Downloaded binaries of third-party frameworks which do not support building with `gyp`.

Keeping Git Submodules Up to Date

The Electron repository has a few vendored dependencies, found in the /vendor directory. Occasionally you might see a message like this when running `git status`:

```
$ git status

    modified:   vendor/brightray (new commits)
    modified:   vendor/node (new commits)
```

To update these vendored dependencies, run the following command:

```
git submodule update --init --recursive
```

If you find yourself running this command often, you can create an alias for it in your `~/.gitconfig` file:

```
[alias]
    su = submodule update --init --recursive
```

Technical Differences Between Electron and NW.js (formerly node-webkit)

Note: Electron was previously named Atom Shell.

Like NW.js, Electron provides a platform to write desktop applications with JavaScript and HTML and has Node integration to grant access to the low level system from web pages.

But there are also fundamental differences between the two projects that make Electron a completely separate product from NW.js:

1. Entry of Application

In NW.js the main entry point of an application is a web page. You specify a main page URL in the `package.json` and it is opened in a browser window as the application's main window.

In Electron, the entry point is a JavaScript script. Instead of providing a URL directly, you manually create a browser window and load an HTML file using the API. You also need to listen to window events to decide when to quit the application.

Electron works more like the Node.js runtime. Electron's APIs are lower level so you can use it for browser testing in place of PhantomJS.

2. Build System

In order to avoid the complexity of building all of Chromium, Electron uses `libchromiumcontent` to access Chromium's Content API. `libchromiumcontent` is a single shared library that includes the Chromium Content module and all of its dependencies. Users don't need a powerful machine to build Electron.

3. Node Integration

In NW.js, the Node integration in web pages requires patching Chromium to work, while in Electron we chose a different way to integrate the libuv loop with each platform's message loop to avoid hacking Chromium. See the `node_bindings` code for how that was done.

4. Multi-context

If you are an experienced NW.js user, you should be familiar with the concept of Node context and web context. These concepts were invented because of how NW.js was implemented.

By using the multi-context feature of Node, Electron doesn't introduce a new JavaScript context in web pages.

Note: NW.js has optionally supported multi-context since 0.13.

Build System Overview

Electron uses gyp for project generation and ninja for building. Project configurations can be found in the `.gyp` and `.gypi` files.

Gyp Files

Following `gyp` files contain the main rules for building Electron:

- `electron.gyp` defines how Electron itself is built.
- `common.gypi` adjusts the build configurations of Node to make it build together with Chromium.
- `vendor/brightray/brightray.gyp` defines how `brightray` is built and includes the default configurations for linking with Chromium.
- `vendor/brightray/brightray.gypi` includes general build configurations about building.

Component Build

Since Chromium is quite a large project, the final linking stage can take quite a few minutes, which makes it hard for development. In order to solve this, Chromium introduced the "component build", which builds each component as a separate shared library, making linking very quick but sacrificing file size and performance.

In Electron we took a very similar approach: for `Debug` builds, the binary will be linked to a shared library version of Chromium's components to achieve fast linking time; for `Release` builds, the binary will be linked to the static library versions, so we can have the best possible binary size and performance.

Minimal Bootstrapping

All of Chromium's prebuilt binaries (`libchromiumcontent`) are downloaded when running the bootstrap script. By default both static libraries and shared libraries will be downloaded and the final size should be between 800MB and 2GB depending on the platform.

By default, `libchromiumcontent` is downloaded from Amazon Web Services. If the `LIBCHROMIUMCONTENT_MIRROR` environment variable is set, the bootstrap script will download from it. `libchromiumcontent-qiniu-mirror` is a mirror for `libchromiumcontent` . If you have trouble in accessing AWS, you can switch the download address to it via `export LIBCHROMIUMCONTENT_MIRROR=http://7xk3d2.dl1.z0.glb.clouddn.com/`

If you only want to build Electron quickly for testing or development, you can download just the shared library versions by passing the `--dev` parameter:

```
$ ./script/bootstrap.py --dev
$ ./script/build.py -c D
```

Two-Phase Project Generation

Electron links with different sets of libraries in `Release` and `Debug` builds. `gyp` , however, doesn't support configuring different link settings for different configurations.

To work around this Electron uses a `gyp` variable `libchromiumcontent_component` to control which link settings to use and only generates one target when running `gyp` .

Target Names

Unlike most projects that use `Release` and `Debug` as target names, Electron uses `R` and `D` instead. This is because `gyp` randomly crashes if there is only one `Release` or `Debug` build configuration defined, and Electron only has to generate one target at a time as stated above.

This only affects developers, if you are just building Electron for rebranding you are not affected.

Tests

Test your changes conform to the project coding style using:

```
$ npm run lint
```

Test functionality using:

```
$ npm test
```

Whenever you make changes to Electron source code, you'll need to re-run the build before the tests:

```
$ npm run build && npm test
```

You can make the test suite run faster by isolating the specific test or block you're currently working on using Mocha's exclusive tests feature. Just append `.only` to any `describe` or `it` function call:

```
describe.only('some feature', function () {
  // ... only tests in this block will be run
})
```

Alternatively, you can use mocha's `grep` option to only run tests matching the given regular expression pattern:

```
$ npm test -- --grep child_process
```

Tests that include native modules (e.g. `runas`) can't be executed with the debug build (see #2558 for details), but they will work with the release build.

To run the tests with the release build use:

```
$ npm test -- -R
```

Build Instructions (macOS)

Follow the guidelines below for building Electron on macOS.

Prerequisites

- macOS >= 10.11.6
- Xcode >= 8.2.1
- node.js (external)

If you are using the Python downloaded by Homebrew, you also need to install the following Python modules:

- pyobjc

macOS SDK

If you're simply developing Electron and don't plan to redistribute your custom Electron build, you may skip this section.

For certain features (e.g. pinch-zoom) to work properly, you must target the macOS 10.10 SDK.

Official Electron builds are built with Xcode 8.2.1, which does not contain the 10.10 SDK by default. To obtain it, first download and mount the Xcode 6.4 DMG.

Then, assuming that the Xcode 6.4 DMG has been mounted at `/Volumes/Xcode` and that your Xcode 8.2.1 install is at `/Applications/Xcode.app` , run:

```
cp -r /Volumes/Xcode/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.10.sdk /Applicati
ons/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/
```

You will also need to enable Xcode to build against the 10.10 SDK:

- Open `/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Info.plist`
- Set the `MinimumSDKVersion` to `10.10`
- Save the file

Getting the Code

```
$ git clone https://github.com/electron/electron
```

Bootstrapping

The bootstrap script will download all necessary build dependencies and create the build project files. Notice that we're using ninja to build Electron so there is no Xcode project generated.

```
$ cd electron
$ ./script/bootstrap.py -v
```

Building

Build both `Release` and `Debug` targets:

```
$ ./script/build.py
```

You can also only build the `Debug` target:

```
$ ./script/build.py -c D
```

After building is done, you can find `Electron.app` under `out/D`.

32bit Support

Electron can only be built for a 64bit target on macOS and there is no plan to support 32bit macOS in the future.

Cleaning

To clean the build files:

```
$ npm run clean
```

Tests

See Build System Overview: Tests

Build Instructions (Windows)

Follow the guidelines below for building Electron on Windows.

Prerequisites

- Windows 7 / Server 2008 R2 or higher
- Visual Studio 2015 Update 3 - download VS 2015 Community Edition for free
- Python 2.7
- Node.js
- Git

If you don't currently have a Windows installation, dev.microsoftedge.com has timebombed versions of Windows that you can use to build Electron.

Building Electron is done entirely with command-line scripts and cannot be done with Visual Studio. You can develop Electron with any editor but support for building with Visual Studio will come in the future.

Note: Even though Visual Studio is not used for building, it's still **required** because we need the build toolchains it provides.

Getting the Code

```
$ git clone https://github.com/electron/electron.git
```

Bootstrapping

The bootstrap script will download all necessary build dependencies and create the build project files. Notice that we're using `ninja` to build Electron so there is no Visual Studio project generated.

```
$ cd electron
$ python script\bootstrap.py -v
```

Building

Build both Release and Debug targets:

```
$ python script\build.py
```

You can also only build the Debug target:

```
$ python script\build.py -c D
```

After building is done, you can find `electron.exe` under `out\D` (debug target) or under `out\R` (release target).

32bit Build

To build for the 32bit target, you need to pass `--target_arch=ia32` when running the bootstrap script:

```
$ python script\bootstrap.py -v --target_arch=ia32
```

The other building steps are exactly the same.

Visual Studio project

To generate a Visual Studio project, you can pass the `--msvs` parameter:

```
$ python script\bootstrap.py --msvs
```

Cleaning

To clean the build files:

```
$ npm run clean
```

Tests

See Build System Overview: Tests

Troubleshooting

Command xxxx not found

If you encountered an error like `Command xxxx not found`, you may try to use the `VS2015 Command Prompt` console to execute the build scripts.

Fatal internal compiler error: C1001

Make sure you have the latest Visual Studio update installed.

Assertion failed: ((handle))->activecnt >= 0

If building under Cygwin, you may see `bootstrap.py` failed with following error:

```
Assertion failed: ((handle))->activecnt >= 0, file src\win\pipe.c, line 1430

Traceback (most recent call last):
  File "script/bootstrap.py", line 87, in <module>
    sys.exit(main())
  File "script/bootstrap.py", line 22, in main
    update_node_modules('.')
  File "script/bootstrap.py", line 56, in update_node_modules
    execute([NPM, 'install'])
  File "/home/zcbenz/codes/raven/script/lib/util.py", line 118, in execute
    raise e
subprocess.CalledProcessError: Command '['npm.cmd', 'install']' returned non-zero exit status 3
```

This is caused by a bug when using Cygwin Python and Win32 Node together. The solution is to use the Win32 Python to execute the bootstrap script (assuming you have installed Python under `C:\Python27`):

```
$ /cygdrive/c/Python27/python.exe script/bootstrap.py
```

LNK1181: cannot open input file 'kernel32.lib'

Try reinstalling 32bit Node.js.

Error: ENOENT, stat 'C:\Users\USERNAME\AppData\Roaming\npm'

Simply making that directory should fix the problem:

```
$ mkdir ~\AppData\Roaming\npm
```

node-gyp is not recognized as an internal or external command

You may get this error if you are using Git Bash for building, you should use PowerShell or VS2015 Command Prompt instead.

Build Instructions (Linux)

Follow the guidelines below for building Electron on Linux.

Prerequisites

- At least 25GB disk space and 8GB RAM.
- Python 2.7.x. Some distributions like CentOS 6.x still use Python 2.6.x so you may need to check your Python version with `python -V`.
- Node.js. There are various ways to install Node. You can download source code from Node.js and compile from source. Doing so permits installing Node on your own home directory as a standard user. Or try repositories such as NodeSource.
- Clang 3.4 or later.
- Development headers of GTK+ and libnotify.

On Ubuntu, install the following libraries:

```
$ sudo apt-get install build-essential clang libdbus-1-dev libgtk2.0-dev \
                       libnotify-dev libgnome-keyring-dev libgconf2-dev \
                       libasound2-dev libcap-dev libcups2-dev libxtst-dev \
                       libxss1 libnss3-dev gcc-multilib g++-multilib curl \
                       gperf bison
```

On RHEL / CentOS, install the following libraries:

```
$ sudo yum install clang dbus-devel gtk2-devel libnotify-devel \
                   libgnome-keyring-devel xorg-x11-server-utils libcap-devel \
                   cups-devel libXtst-devel alsa-lib-devel libXrandr-devel \
                   GConf2-devel nss-devel
```

On Fedora, install the following libraries:

```
$ sudo dnf install clang dbus-devel gtk2-devel libnotify-devel \
                   libgnome-keyring-devel xorg-x11-server-utils libcap-devel \
                   cups-devel libXtst-devel alsa-lib-devel libXrandr-devel \
                   GConf2-devel nss-devel
```

Other distributions may offer similar packages for installation via package managers such as pacman. Or one can compile from source code.

Getting the Code

```
$ git clone https://github.com/electron/electron.git
```

Bootstrapping

The bootstrap script will download all necessary build dependencies and create the build project files. You must have Python 2.7.x for the script to succeed. Downloading certain files can take a long time. Notice that we are using `ninja` to build Electron so there is no `Makefile` generated.

```
$ cd electron
$ ./script/bootstrap.py -v
```

Cross compilation

If you want to build for an `arm` target you should also install the following dependencies:

```
$ sudo apt-get install libc6-dev-armhf-cross linux-libc-dev-armhf-cross \
                       g++-arm-linux-gnueabihf
```

And to cross compile for `arm` or `ia32` targets, you should pass the `--target_arch` parameter to the `bootstrap.py` script:

```
$ ./script/bootstrap.py -v --target_arch=arm
```

Building

If you would like to build both `Release` and `Debug` targets:

```
$ ./script/build.py
```

This script will cause a very large Electron executable to be placed in the directory `out/R`. The file size is in excess of 1.3 gigabytes. This happens because the Release target binary contains debugging symbols. To reduce the file size, run the `create-dist.py` script:

```
$ ./script/create-dist.py
```

This will put a working distribution with much smaller file sizes in the `dist` directory. After running the create-dist.py script, you may want to remove the 1.3+ gigabyte binary which is still in `out/R`.

You can also build the `Debug` target only:

```
$ ./script/build.py -c D
```

After building is done, you can find the `electron` debug binary under `out/D`.

Cleaning

To clean the build files:

```
$ npm run clean
```

Troubleshooting

Error While Loading Shared Libraries: libtinfo.so.5

Prebulit `clang` will try to link to `libtinfo.so.5`. Depending on the host architecture, symlink to appropriate `libncurses`:

```
$ sudo ln -s /usr/lib/libncurses.so.5 /usr/lib/libtinfo.so.5
```

Tests

See Build System Overview: Tests

Advanced topics

The default building configuration is targeted for major desktop Linux distributions, to build for a specific distribution or device, following information may help you.

Building `libchromiumcontent` locally

To avoid using the prebuilt binaries of `libchromiumcontent`, you can pass the `--build_libchromiumcontent` switch to `bootstrap.py` script:

```
$ ./script/bootstrap.py -v --build_libchromiumcontent
```

Note that by default the `shared_library` configuration is not built, so you can only build `Release` version of Electron if you use this mode:

```
$ ./script/build.py -c R
```

Using system `clang` instead of downloaded `clang` binaries

By default Electron is built with prebuilt `clang` binaries provided by Chromium project. If for some reason you want to build with the `clang` installed in your system, you can call `bootstrap.py` with `--clang_dir=<path>` switch. By passing it the build script will assume the `clang` binaries reside in `<path>/bin/`.

For example if you installed `clang` under `/user/local/bin/clang`:

```
$ ./script/bootstrap.py -v --build_libchromiumcontent --clang_dir /usr/local
$ ./script/build.py -c R
```

Using other compilers other than `clang`

To build Electron with compilers like `g++`, you first need to disable `clang` with `--disable_clang` switch first, and then set `CC` and `CXX` environment variables to the ones you want.

For example building with GCC toolchain:

```
$ env CC=gcc CXX=g++ ./script/bootstrap.py -v --build_libchromiumcontent --disable_clang
$ ./script/build.py -c R
```

Environment variables

Apart from `CC` and `CXX`, you can also set following environment variables to custom the building configurations:

- `CPPFLAGS`
- `CPPFLAGS_host`
- `CFLAGS`
- `CFLAGS_host`
- `CXXFLAGS`
- `CXXFLAGS_host`
- `AR`

- `AR_host`
- `CC`
- `CC_host`
- `CXX`
- `CXX_host`
- `LDFLAGS`

The environment variables have to be set when executing the `bootstrap.py` script, it won't work in the `build.py` script.

Debugging on Windows

If you experience crashes or issues in Electron that you believe are not caused by your JavaScript application, but instead by Electron itself, debugging can be a little bit tricky, especially for developers not used to native/C++ debugging. However, using Visual Studio, GitHub's hosted Electron Symbol Server, and the Electron source code, it is fairly easy to enable step-through debugging with breakpoints inside Electron's source code.

Requirements

- **A debug build of Electron**: The easiest way is usually building it yourself, using the tools and prerequisites listed in the build instructions for Windows. While you can easily attach to and debug Electron as you can download it directly, you will find that it is heavily optimized, making debugging substantially more difficult: The debugger will not be able to show you the content of all variables and the execution path can seem strange because of inlining, tail calls, and other compiler optimizations.
- **Visual Studio with C++ Tools**: The free community editions of Visual Studio 2013 and Visual Studio 2015 both work. Once installed, configure Visual Studio to use GitHub's Electron Symbol server. It will enable Visual Studio to gain a better understanding of what happens inside Electron, making it easier to present variables in a human-readable format.
- **ProcMon**: The free SysInternals tool allows you to inspect a processes parameters, file handles, and registry operations.

Attaching to and Debugging Electron

To start a debugging session, open up PowerShell/CMD and execute your debug build of Electron, using the application to open as a parameter.

```
$ ./out/D/electron.exe ~/my-electron-app/
```

Setting Breakpoints

Then, open up Visual Studio. Electron is not built with Visual Studio and hence does not contain a project file - you can however open up the source code files "As File", meaning that Visual Studio will open them up by themselves. You can still set breakpoints - Visual Studio will automatically figure out that the source code matches the code running in the attached process and break accordingly.

Relevant code files can be found in `./atom/` as well as in Brightray, found in `./vendor/brightray/browser` and `./vendor/brightray/common`. If you're hardcore, you can also debug Chromium directly, which is obviously found in `chromium_src`.

Attaching

You can attach the Visual Studio debugger to a running process on a local or remote computer. After the process is running, click Debug / Attach to Process (or press `CTRL+ALT+P`) to open the "Attach to Process" dialog box. You can use this capability to debug apps that are running on a local or remote computer, debug multiple processes simultaneously.

If Electron is running under a different user account, select the `Show processes from all users` check box. Notice that depending on how many BrowserWindows your app opened, you will see multiple processes. A typical one-window app will result in Visual Studio presenting you with two `Electron.exe` entries - one for the main process and one for the renderer process. Since the list only gives you names, there's currently no reliable way of figuring out which is which.

Which Process Should I Attach to?

Code executed within the main process (that is, code found in or eventually run by your main JavaScript file) as well as code called using the remote (`require('electron').remote`) will run inside the main process, while other code will execute inside its respective renderer process.

You can be attached to multiple programs when you are debugging, but only one program is active in the debugger at any time. You can set the active program in the `Debug Location` toolbar or the `Processes window` .

Using ProcMon to Observe a Process

While Visual Studio is fantastic for inspecting specific code paths, ProcMon's strength is really in observing everything your application is doing with the operating system - it captures File, Registry, Network, Process, and Profiling details of processes. It attempts to log **all** events occurring and can be quite overwhelming, but if you seek to understand what and how your application is doing to the operating system, it can be a valuable resource.

For an introduction to ProcMon's basic and advanced debugging features, go check out this video tutorial provided by Microsoft.

Setting Up Symbol Server in Debugger

Debug symbols allow you to have better debugging sessions. They have information about the functions contained in executables and dynamic libraries and provide you with information to get clean call stacks. A Symbol Server allows the debugger to load the correct symbols, binaries and sources automatically without forcing users to download large debugging files. The server functions like Microsoft's symbol server so the documentation there can be useful.

Note that because released Electron builds are heavily optimized, debugging is not always easy. The debugger will not be able to show you the content of all variables and the execution path can seem strange because of inlining, tail calls, and other compiler optimizations. The only workaround is to build an unoptimized local build.

The official symbol server URL for Electron is https://electron-symbols.githubapp.com. You cannot visit this URL directly, you must add it to the symbol path of your debugging tool. In the examples below, a local cache directory is used to avoid repeatedly fetching the PDB from the server. Replace `c:\code\symbols` with an appropriate cache directory on your machine.

Using the Symbol Server in Windbg

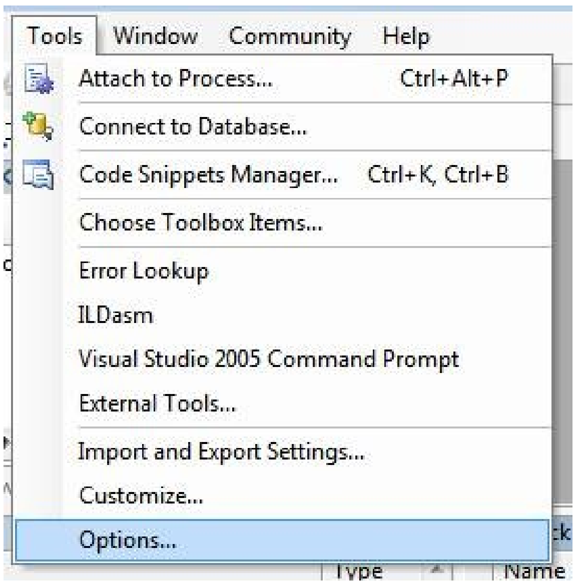
The Windbg symbol path is configured with a string value delimited with asterisk characters. To use only the Electron symbol server, add the following entry to your symbol path (**Note:** you can replace `c:\code\symbols` with any writable directory on your computer, if you'd prefer a different location for downloaded symbols):

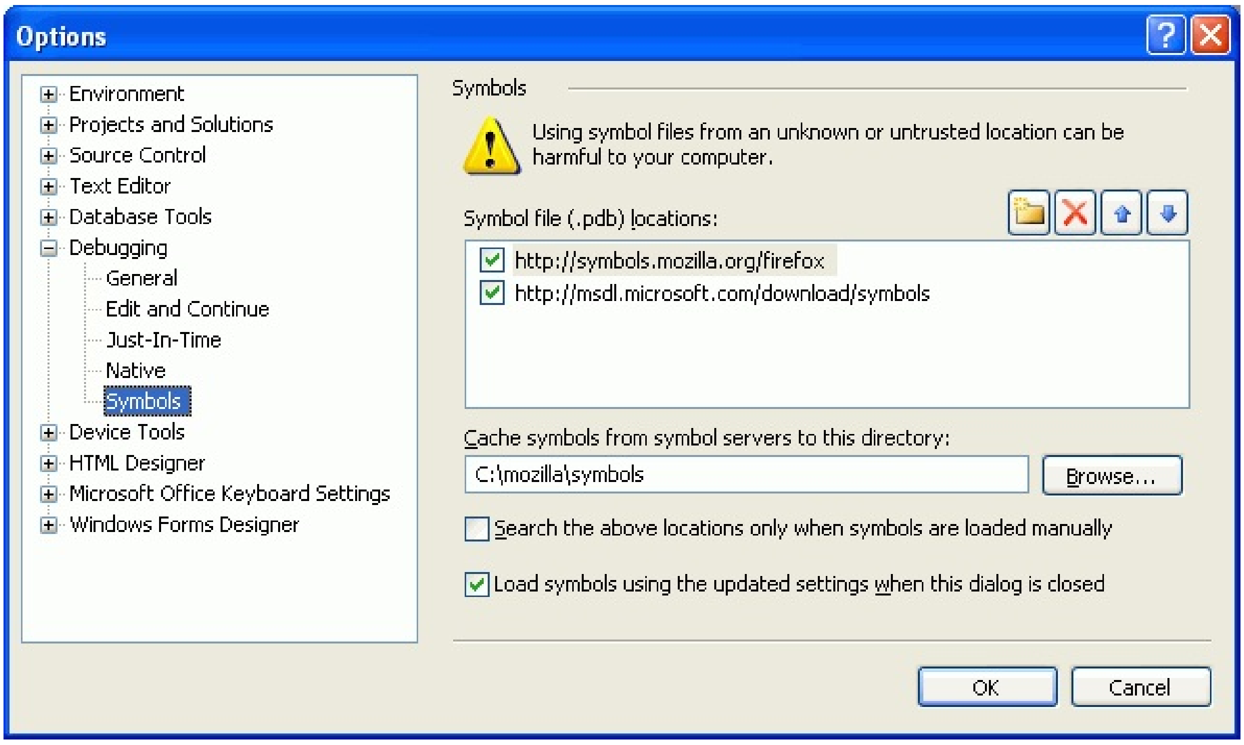
```
SRV*c:\code\symbols\*https://electron-symbols.githubapp.com
```

Set this string as `_NT_SYMBOL_PATH` in the environment, using the Windbg menus, or by typing the `.sympath` command. If you would like to get symbols from Microsoft's symbol server as well, you should list that first:

```
SRV*c:\code\symbols\*http://msdl.microsoft.com/download/symbols;SRV*c:\code\symbols\*https://electron-symbols.githuba
pp.com
```

Using the symbol server in Visual Studio


Tools
Window
Community
Help
Attach to Process... Ctrl+Alt+P
Connect to Database...
Code Snippets Manager... Ctrl+K, Ctrl+B
Choose Toolbox Items...
Error Lookup
ILDasm
Visual Studio 2005 Command Prompt
External Tools...
Import and Export Settings...
Customize...
Options...



Symbols

Using symbol files from an unknown or untrusted location can be harmful to your computer.

Symbol file (.pdb) locations:

- [✓] http://symbols.mozilla.org/firefox
- [✓] http://msdl.microsoft.com/download/symbols

Cache symbols from symbol servers to this directory: C:\mozilla\symbols

- [ ] Search the above locations only when symbols are loaded manually
- [✓] Load symbols using the updated settings when this dialog is closed

Troubleshooting: Symbols will not load

Type the following commands in Windbg to print why symbols are not loading:

```
> !sym noisy
> .reload /f electron.exe
```