Native Node Modules

Native Node.js modules are supported by Electron, but since Electron has a different application binary interface (ABI) from a given Node.js binary (due to differences such as using Chromium's BoringSSL instead of OpenSSL), the native modules you use will need to be recompiled for Electron. Otherwise, you will get the following class of error when you try to run your app:

```
Error: The module '/path/to/native/module.node'
was compiled against a different Node.js version using
NODE_MODULE_VERSION $XYZ. This version of Node.js requires
NODE_MODULE_VERSION $ABC. Please try re-compiling or re-installing
the module (for instance, using `npm rebuild` or `npm install`).
```

How to install native modules

There are several different ways to install native modules:

Installing modules and rebuilding for Electron

You can install modules like other Node projects, and then rebuild the modules for Electron with the `electron-rebuild` package. This module can automatically determine the version of Electron and handle the manual steps of downloading headers and rebuilding native modules for your app. If you are using Electron Forge, this tool is used automatically in both development mode and when making distributables.

For example, to install the standalone `electron-rebuild` tool and then rebuild modules with it via the command line:

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

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

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

For more information on usage and integration with other tools such as Electron Packager, consult the project's README.

Using `npm`

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

For example, to install all dependencies for Electron:

```
# Electron's version.
export npm_config_target=1.2.3
```

```
# The architecture of Electron, see https://electronjs.org/docs/tutorial/support#supported-p
# for supported architectures.
export npm_config_arch=x64
export npm_config_target_arch=x64
# Download headers for Electron.
export npm_config_disturl=https://electronjs.org/headers
# 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
```

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://electron
```

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

Manually building for a custom build of Electron

To compile native Node modules against a custom build of Electron that doesn't match a public release, instruct `npm` to use the version of Node you have bundled with your custom build.

```
npm rebuild --nodedir=/path/to/src/out/Default/gen/node_headers
```

Troubleshooting

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

- When in doubt, run `electron-rebuild` first.
- Make sure the native module is compatible with the target platform and architecture for your Electron app.
- Make sure `win_delay_load_hook` is not set to `false` in the module's `binding.gyp`.
- After you upgrade Electron, you usually need to rebuild the modules.

A note about `win_delay_load_hook`

On Windows, by default, `node-gyp` links native modules against `node.dll`. However, in Electron 4.x and higher, the symbols needed by native modules are exported by `electron.exe`, and there is no `node.dll`. In order to load native modules on Windows, `node-gyp` installs a delay-load hook that triggers when the native module is loaded, and redirects the `node.dll` reference to use the loading executable instead of looking for `node.dll` in the library search path (which would turn up nothing). As such, on Electron 4.x and higher, `'win_delay_load_hook': 'true'` is required to load native modules.

If you get an error like `Module did not self-register`, or `The specified procedure could not be found`, it may mean that the module you're trying to use did not correctly include the delay-load hook. If the module is built with node-gyp, ensure that the `win_delay_load_hook` variable is set to `true` in the `binding.gyp` file, and isn't getting overridden anywhere. If the module is built with another system, you'll need to ensure that you build with a delay-load hook installed in the main `.node` file. Your `link.exe` invocation should look like this:

```
link.exe /OUT:"foo.node" "...\node.lib" delayimp.lib /DELAYLOAD:node.exe /DLL
    "my_addon.obj" "win_delay_load_hook.obj"
```

In particular, it's important that:

- you link against `node.lib` from *Electron* and not Node. If you link against the wrong `node.lib` you will get load-time errors when you require the module in Electron.
- you include the flag `/DELAYLOAD:node.exe`. If the `node.exe` link is not delayed, then the delay-load hook won't get a chance to fire and the node symbols won't be correctly resolved.
- `win_delay_load_hook.obj` is linked directly into the final DLL. If the hook is set up in a dependent DLL, it won't fire at the right time.

See `node-gyp` for an example delay-load hook if you're implementing your own.

Modules that rely on `prebuild`

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

If the `prebuild`-powered module 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.

Sometimes those modules work fine under Electron, but when there are no Electron-specific binaries available, you'll need to build from source. Because of this, it is recommended to use `electron-rebuild` for these modules.

If you are following the `npm` way of installing modules, you'll need to pass `--build-from-source` to `npm`, or set the `npm_config_build_from_source` environment variable.