西南交通大学学报

第 54 卷 第 4 期
2019 年 8 月

JOURNAL OF SOUTHWEST JIAOTONG UNIVERSITY

Vol. 54 No.4
Aug. 2019

ISSN - 0258-2724 DOI：10.35741/issn.0258-2724.54.4.1

Research article

Computer Science

AN IN-DEPTH COMPARISON OF SOFTWARE FRAMEWORKS FOR DEVELOPING DESKTOP APPLICATIONS USING WEB TECHNOLOGIES

Zinah Hussein Toman[a], Sarah Hussein Toman[b], Manar Joundy Hazar[c]

[a] Computer Science Department, College of Computer Science and Information Technology, University of Al -QadisiyahAd-Diwaniah, Iraq
E-mail: Zinah.Hussein@qu.edu.iq

[b] Roads and Transport Department, College of Engineering, University of Al –Qadisiyah, Ad-Diwaniah, Iraq
E-mail: sarah.toman@qu.edu.iq

[c] Computer Center, University of Al –Qadisiyah, Ad-Diwaniah, Iraq
E-mail: manar.joundy@qu.edu.iq

Abstract

Today JavaScript is one of the most popular and fastest growing programming languages. Initially designed as a web browser scripting language, its adoption has reached beyond web pages: the Internet of Things, mobile and desktop applications. Lately, an increased interest can be observed over developing desktop software using JavaScript and other web technologies such as HTML and CSS. Many popular software products followed this path: Skype, Visual Studio Code, Atom, Brackets, Light Table, Microsoft Teams, Microsoft SQL Operations Studio, GitHub Desktop, Signal, etc.

The aim of this article is to aid developers to choose the right framework for their needs, through a comprehensive side-by-side comparison of Electron and NW.js, the two frameworks available for developing desktop software with JavaScript, HTML and CSS. Electron despite being a younger project. It was concluded in this article that this software framework outperforms NW.js in the matter of capabilities on most areas such as file system, user interface, system integration and multimedia; except printing. However, NW.js is easier to use and debug.

Keywords: JavaScript, Cross-platform, Desktop, Framework, Electron, NW.js.

摘要：今天，JavaScript 是最受欢迎和发展最快的编程语言之一。最初设计为 Web 浏览器脚本语言，其应用已超越网页：物联网，移动和桌面应用程序。最近，使用 JavaScript 和其他 Web 技术（如 HTML 和 CSS）开发桌面软件时，可以观察到越来越多的兴趣。许多流行的软件产品都遵循这条道路：Skype，Visual Studio Code，Atom，Brackets，Light Table，Microsoft Teams，Microsoft SQL Operations Studio，GitHub Desktop，Signal 等。本文的目的是通过对 Electron 和 NW.js 的全面并排比较，帮助开发人员为他们的需求选择合适的框架，这两个框架可用于使用 JavaScript，HTML 和 CSS 开发桌面软件.Electron 尽管是一个年轻的项目。本文的结论是，在文件系统，用户界面，系统集成和多媒体等大多数领域的能力方面，它优于 NW.js;除了印刷。但是，NW.js 更易于使用和调试

关键词: JavaScript，跨平台，桌面，框架，电子

I. INTRODUCTION

Developers seeking to build a desktop application using web technologies have two options in the matter of software frameworks: Electron and NW.js [1].

Electron and NW.jsare open-source frameworks that provide an infrastructure and API to allow web applications to run as native applications, taking advantage of advanced features otherwise not available for web applications [1][2][3].

Electron and NW.js can be used to build Windows, MacOS and Linux GUI native applications [5][8][9], using the same JavaScript, HTML and CSS codebase between all platforms [1]. Electron also supports Microsoft Store and Mac Store packaged apps. Their architecture is similar: Chromium's engine and Node.js are merged together into a single runtime [1].

Chromium's engine – the same as behind the Google Chrome web browser, has the role of rendering HTML and CSS into native windows. Thus, applications based on Electron and NW.js take advantage of the latest HTML and CSS features.

Node.js is a JavaScript runtime based on the V8 engine, used for command line and server-side scripting [4][5]. It includes built-in support for file system access, binary data (buffers) manipulation, networking, cryptography, process-related functions, and other core features. Node.js benefits of a large ecosystem of third-party modules (libraries) [6][7]. As of writing this paper, NPM, the largest Node.js module repository, was containing 857,290 modules.

The Chromium's engine and Node.js pair build a web browser-like environment for web applications, but without usual sandboxing that prevents the scripting from interoperating with the local system [1].

However, Electron and NW.js feature significant differences in the matter of capabilities, as will be further detailed. This comparison Electron 3.0 and NW.js 0.33.3 were evaluated.

II. THE COMPARISON

A. Architecture

Both Electron and NW.js frameworks are distributed with a precompiled native executable file and support libraries. Developers need only to include some configuration files and application files (codebase and assets).

To create an NW.js application, the developer should create a "manifest.json" configuration file on the root directory of the application pointing to the HTML page that should be loaded into the initial window, along with other window and runtime settings. Alternatively, a script file can be used to initiate the main window.

Developers using Electron framework should include their application files in resources/app subdirectory. Here a "package.json" file should be created pointing to a script file that will initialize the application. This script file will be executed in the so-called "main process".

Like NW.js, Electron features a multi-process architecture. Electron has an architecture particularity, having 2 types of processes:

- ***renderer process*** - a process running the JavaScript engine instance associated to each window.
- ***main process*** - a process containing the JavaScript engine instance used to execute the JavaScript file specified in "package.json". Unlike a renderer process, this process runs in Node.js-only environment, without access to DOM or Web Browser APIs. Its purpose is to create the main window and manage renderer processes. There may be just one main process per application instance.

In Electron it is recommended to call GUI-related APIs only from the main process to avoid memory leaks. Scripts from renderer processes should communicate with the script from the main process in order to request GUI operations. Some Electron APIs are available only in the main process.

The object exposing Electron's API is retrieved in JavaScript by calling require ("electron") for the main process and require ("electron") remotefor renderer processes. NW.js exposes its API vianw global variable.

Electron can package the files of the web application, its assets and Node modules into a single file called ASAR.

NW js can use compiled JavaScript (V8 snapshots) files instead of plain-text, although the performance is reduced compared to compiling JavaScript code at runtime.

B. User Interface

Analogous to web applications, applications built with Electron or NW.js use exclusively HTML and CSS to define their user interface.

Basic user interface elements like buttons, checkboxes, radio buttons, text boxes, combo boxes, sliders, progress bars, etc, that are supported by default in HTML are available. The design of these elements is identic on Electron and NW.js, sharing exactly the same design like

in Chromium web browser. On MacOS their look is similar to the native counterparts, but on Windows and Linux their design is completely unrelated to the native user interface elements. Thus, Electron and NW.js cannot provide a user interface consistent with the current operating system's native look on Windows and Linux.

Electron and NW.js renders a web page into its individual native window, instead of rendering into a tab located on the web browser's window like in the case of a web application. There is programmatic access to control common aspects of the window such as the title, title bar buttons, icon, size and position.

Placing web contents into native windows represent the main user interface adaptation of web technologies towards desktop. The window features among Electron and NW.js varies, as detailed in the following tables which perform a comprehensive comparison.

Table 1.
A comprehensive comparison of window features support [8][9]

Feature	Electron	NW.js
Always on top	✓	✓
Frameless	✓	✓
Drag area for frameless windows	✓	✓
Define title bar type (Mac)	✓	-
Flash frame	✓	✓
Thick Frame (Windows)	✓	-
Minimum and maximum size limit	✓	✓
Disable minimize, maximize and close buttons	✓	-
Disable the ability to resize	✓	✓
Auto-size to contents	✓	-
Adjust zoom level	✓	✓
Zoom to page width	✓	✓
Animate position change (MacOS)	✓	-
Disable the ability to move	✓	-
Allow the window to be larger than the screen	✓	-
Hide icon from taskbar	✓	✓
Overlay icon in taskbar	✓	-
Progress bar in taskbar	✓	✓*
Custom crop area for taskbar thumbnail	✓	-
Taskbar thumb buttons (Windows)	✓	-
Badge (Ubuntu)	-	✓
Set the background color shown before the content is loaded	✓	-
Set opacity	✓	-
Use transparency	✓	✓**
Auto-hide menu bar	✓	-
Page Visibility Events	✓	-
Hidden until explicitly shown	✓	✓
Off-screen (headless) rendering	✓	-
Disable the ability to be focused	✓	-
Ability to enter in fullscreen (MacOS)	✓	-
Simple fullscreen mode (MacOS)	✓	-
Kiosk Mode	✓	✓
Enable dark theme (Linux with GTK3 only)	✓	-
Define Vibrancy (MacOS)	✓	-
Modal sheet offset (MacOS)	✓	-
Show and hide the shadow (MacOS)	✓	✓
Fullscreen title (MacOS)	✓	-
Represented file name (MacOS)	✓	-
Document edited indicator (MacOS)	✓	-
Tabbing identifier (MacOS)	✓	-
Disable cursor auto-hiding on input fields	✓	-
Hook window message (Windows)	✓	-

*Note: * Available on Ubuntu Linux only, ** Deprecated*

Each window of an application has its separate instance of renderer and JavaScript engine. Splitting the user interface into multiple native windows and dialogs is less desirable, due to a significant penalty in the matter of memory and CPU cycles usage.

Electron allows the creation of modal windows also. On MacOS dialog windows become modal sheets. There is no support in NW.js for creating modal dialog windows.

On Electron the first window is created from the main process via JavaScript. Subsequent windows may be created from an already created window or the main process.

NW.js can create its first window according to the settings defined in "manifest.json" file. Creating the initial window programmatically from JavaScript is also possible.

Table 2.
Support level for observable events on windows

Event	Electron	NW.js
Maximize, unmaximize, minimize, restore	✓	✓
Resize	✓	✓
Move	✓	✓
Title change	✓	-
Zoom	-	✓
Enter and leave full screen	✓	✓
Child opening	-	✓
Keyboard multimedia keys	✓	-
Scroll touch (Mac)	✓	-
3 fingers swipe (Mac)	✓	-
Navigating to another document	-	✓

Table 3.
Window style support comparison [8][9]

Style	Electron	NW.js
Windows		
Toolbar	✓	-
Linux		
Desktop	✓	✓
Dock	✓	-
Toolbar	✓	-

Splash	✓	-
Notification	✓	-
MacOS		
Desktop	✓	-
Textured	✓	-

Since each window has its own individual JavaScript engine instance, sharing data between windows is limited. Objects passed between windows are serialized via a structured clone algorithm on both frameworks. NW.js use the standard `postMessage`JavaScript method to send data to another window, while Electron uses its own implementation provided by the `ipcRenderer`object from electron module. Electron windows may also communicate with the main process.

Different common dialogs are available to Electron and NW.js developers.

Alert, confirmation and prompt dialogs, provided by all web browsers are available on NW.js and feature Google's Material Design for their appearance. Electron's alert and confirmation dialogs are using the native dialog of the operating system instead. Electron does not support prompt dialogs, including in the case of displaying a website from Internet that may require it. There are no options to customize the appearance of these dialogs.

Electron provides an additional native message dialog box where title, icons and buttons can be customized. Also may include a checkbox or links.

Both Electron and NW.js applications can invoke file open, file save, and directory open native dialogs, a capability mandatory to desktop apps. NW.js provides access to these dialogs via HTML DOM (`<input type="file">`) while Electron provides it via `dialog`object from Electron-built-in module. Electron provides some extra customizations such as:

- Controlling the security-scoped bookmarks for Mac App Store applications.
- Disabling automatic symlink path resolution;
- Treating .app packages as a directory, instead of a file on MacOS;
- Allowing the user to specify a non-existing file in open dialog on Windows;
- Allowing the creation of directories from directory open dialog on MacOS;
- Customizing text of the default button in save file dialog;
- Customizing the message and file name field label in file save dialog on MacOS;
- Hiding the tags field from file save dialog on MacOS.

A color picker dialog is available also on both frameworks. On Windows and MacOS the operating system's native color picker dialog is shown, while on Linux a custom color picker featuring Google Chrome's user interface design is shown.

Electron is able also to display on MacOS a native "About" dialog with details of the current application and a file preview dialog. On all operating systems an error box and a certificate trust dialog are also available on Electron.

Electron and NW.js can provide a menu bar for each window on Windows and Linux. MacOS by design supports only a single menu bar per application, displayed on the top of the screen. Context menus are supported also. These menus support icons, labels, keyboard shortcut indicators, separators, and checkboxes. Radio menu items and sub-labels are exclusive to Electron while menu item tooltips and icon masks are exclusive to NW.js. It is not possible to customize
the styling of menus. If such styling is required, it can be achieved by replicating these features with HTML and CSS.

Applications may also create a system tray icon associated with a menu on all platforms excepting Linux with Gnome 3 environment where system tray icons are not supported by the system itself. The next table highlights the differences between Electron and NW.js for system tray icon features [8][9].

Table 4.
System tray icon features support comparison

Feature	Electron	NW.js
Tooltip	✓	✓
Message balloon	✓	-
Title (MacOS)	✓	✓
Pressed (alternate) image on Mac	✓	✓
Highlight mode	✓	-
Icon as template	-	✓
Querying bounds on screen	✓	-
Determining the mouse button used to click	✓	-
Determining the keyboard modifiers for click	✓	-
Detecting double click on tray icon	✓	-
Mouse enter/leave events on MacOS	✓	-

Notification boxes are supported by both frameworks and can be invoked using W3C's Web Notification API. Electron also provides access to notifications from the main process. The notification box cannot be restyled.

Both frameworks support registering global keyboard shortcuts that will invoke a callback even if none of the application's window is currently focused.

Observing display parameters is supported also. Electron and NW.js can trigger events when a monitor is added and removed or when the display bounds changes. Electron can also observe scale factor and orientation changes.

Additionally, Electron can query or be notified about changes to system preferences like accent color, dark or light user interface mode, default system colors, high-contrast color scheme presence and Aero Glass availability.

On MacOS, Electron can take advantage of Mac Dock integration, such as: setting the icon, associating a menu, showing and hiding the icon, bouncing the icon and display download progress.

Electron can also interact with the touch bar of the fourth generation of MacBook Pro devices.

C. System Integration

Applications developed using Electron and NW.js frameworks take advantage of access to advanced system features, that are otherwise inaccessible to web applications. These features will be discussed further below.

Reading and writing clipboard contents is a common operation in desktop applications. Electron and NW.js provide access to the clipboard with support for different data formats as enumerated in table 5 [8][9].

Table 5.
Clipboard data format support comparison

Data Format	Electron	NW.js
Plain text	✓	✓
HTML	✓	✓
RTF	✓	✓
Image	✓	✓
Bookmark	✓	-
Custom	✓	-

Both Electron and NW.js are able to spawn and manipulate processes, access their standard streams (stdin, stdout and stderr) through “process” Node.js built-in module. Opening a URL in default browser is also supported by both frameworks.

Electron can additionally:

- Query current locale;
- Register or unregister the current application as a default protocol handler;
- Set the current application to automatically launch after OS boots;
- Return an error code after the application execution ends;
- Define tasks for application's JumpList on Windows;
- Define MacOS activities;
- Set App User Model ID on Windows;
- Query if the current application runs from "App" folder in MacOS;
- Move the current application to "App" folder in MacOS;
- Prevent the system from entering a standby state;
- Provide integration for MacOSInApp Purchase;
- Trigger events for system shutdown/suspend/resume;
- Triggers an event for power source changes (battery or AC);
- Query CPU, memory and I/O statistics.

None of these two frameworks are able to directly access the operating system's API to take advantage of the entire set of native features. Additional features may be accessible through native third-party Node.js modules.

D. Printing

Printing data is a common task for desktop applications, thus the print capabilities of Electron and NW.js are also an important area to discuss. Both have the ability to print HTML documents either to a physical printer or to PDF file.

NW.js features programmatic control over basic printing settings, such as configuring paper size, paper orientation and margins. Electron seems to support these for printing to PDF files, but not for physical printers also. Programmatically setting the copy count and contents scale is supported exclusively by NW.js.

A common difficulty experienced by web developers while printing from client-side consists in adding headers and footers to pages, respectively page numbering. Web browsers provide these capabilities but can be manipulated only by the user. Unfortunately, developers have yet no programmatic way to access them.

While Electron gives no such capabilities, NW.js allows the developer to define a plain text header and footer. The footer will always contain the page number on the right side. If only a page numbering is desired, it can be achieved by setting a blank header and footer.

Before printing, the user has the ability to select a printer, define page size, orientation, and other settings depending on the printer. Electron will display the operating system's print settings dialog, while NW.js will display the Chromium's print dialog that also includes a print preview.

The print dialog of NW.js uses the same user interface design like Chromium, without an ability to restyle it. This print dialog can be replaced with the operating system's print settings dialog by specifying `--disable-print-preview` as Chromium argument.

Both Electron and NW.js are able to enumerate printers and print without user interaction (bypassing any print dialog). However, NW.js will display Chromium's print dialog for short time before automatically closing and starting the print operation.

E. File System

Electron and NW.js feature the same capabilities in the matter of basic file system operations as both include “fs”, a module available by default in Node.js. These capabilities include: enumerating, creating, deleting, querying and changing permission/attributes of files and directories. Files can be open for reading and writing in text or binary mode, supporting both endianness and different text encodings. Additionally, the file system can be watched to observe changes on files and directories caused by other processes.

Both can also easily manipulate file and directory paths indifferent if the path is in a Windows or POSIX form, through “path” module, also a built-in module of Node.js. The system directory paths that can be queried, are enumerated in table 6 [8][9].

Table 6.
System directory paths support comparison

Directory	NW.js	Electron
User	✓*	✓
Per-user application data	-	✓
Application's user data	✓	✓
Temporary directory	✓*	✓
Desktop	-	✓
Documents	-	✓
Pictures	-	✓
Music	-	✓
Videos	-	✓
Downloads	-	✓

Note: * via Node.js

Electron and NW.js can invoke the default associated program to open a certain file or to open the file manager to highlight a file or directory.

Electron can additionally move files/directories to trash, add files to recent document list of the OS on Windows and MacOS, query the icon of files/directories, query current executable path respectively to read and write shortcut files on Windows.

Electron and NW.js applications may benefit of additional file system related capabilities by using third-party Node.js modules. NW.js may compensate for the lack of features through third-party modules also.

F. Multimedia

NW.js has a set of built-in codecs and demuxers: Theora, Vorbis, PCM, MP3, OGG, Matroska, WAV. Additional formats such as H.264 and MP4 are available only by manually rebuilding "ffmpeg.dll" library (for legal reasons). Electron supports all previously mentioned codecs and demuxers, including H.264 and MP4.

Electron has also built-in support for basic image manipulation, like cropping, scaling, HSL shifting and converting formats (PNG, JPG, BMP). Capturing audio and video from the desktop is also supported in Electron.

J. Debugging

Both Electron and NW.js feature the standard Developer Tools available in Chromium. Developer Tools contain a set of powerful elements: JavaScript debugger, DOM element/CSS rule inspector, console to evaluate JavaScript instructions, CPU and memory profiling, etc.

Electron's Developer Tools can be opened for each window programmatically from JavaScript. The script running in the main process can be debugged only by an external debugger, after launching the executable file with `–inspector` or `--inspect-brk` command line argument.

The Developer Tools of NW.js can be opened at any time by right-clicking on the surface of any NW.js application window. NW.js also features a built-in task manager to monitor memory footprint, CPU usage, consumed network bandwidth and process ID of each process used by NW.js runtime. Developer Tools are available only in the SDK build flavor of NW.js.

III. CONCLUSIONS

As it could be observed earlier, Electron proved to be significantly more capable than NW.js in system integration and user interface. Electron also proved to be slightly more capable in file system and multimedia features.

Applications built on NW.js proved to be easier to debug, as Developer Tools can be invoked any time from the context menu available on any window. Developer Tools in Electron can be opened only programmatically

and an external debugger is required to debug the script running in the main process.

Less experienced programmers may find easier to use NW.js, as only creating a "manifest.json" configuration file is needed to wrap a regular web application as a desktop application. No Node.js knowledge is required if the application needs only features available in regular web browsers. Electron introduces additional complexity through initializing the initial window through a script running in the main process, respectively by the limited communication support between main process/renderer processes.

Electron's printing capabilities are inferior compared to NW.js, lacking even basic features such as setting paper size, paper orientation and margins; if the printing destination is other than a PDF file. Electron also lacks support for headers and footers.

The print preview dialog using Chromium's UI design and message/prompt dialogs featuring Google's Material Design may be an obstacle for the developers trying to achieve a consistent user interface design for the applications based on NW.js.

REFERENCES

[1]JENSEN, P.B. (2017). *Cross-platform Desktop Applications: Using Electron and NW. js.* Manning Publications Company.

[2]KREDPATTANAKUL, K., and LIMPIYAKORN, Y. (2019). Transforming JavaScript-Based Web Application to Cross-Platform Desktop with Electron. *Proceedings of the 9th International Conference on Information Science and Applications (ICISA 2018)*. Springer, Singapore, pp. 571-579. DOI: 10.1007/978-981-13-1056-0.

[3]Benoit, A. (2015). *NW. js Essentials.* Packt Publishing Ltd.

[4]DIPIERRO, M. (2018). The Rise of JavaScript. *Computing in Science & Engineering*, 20(1), pp. 9-10.

[5]PARSHINA, M. (2018). *JavaScript beyond the browser*. Bachelor's Thesis in Information Technology. Turku University of Applied Sciences.

[6]MARDAN, A. (2015). Intro to Node. js. Chapter 6. In *Full Stack JavaScript*. Apress, Berkeley, CA, pp. 137-154. DOI: 10.1007/978-1-4842-1751-1

[7]IHRIG, C.J. (2013). The Node Module System. Chapter 2. In *Pro Node. js for Developers*. Apress, Berkeley, CA, pp. 9-27.

[8]Electron Documentation. Guides: Getting Started with Electron. Available from https://electronjs.org/docs

[9]NW.js Documentation for 0.13 and later. Available from http://docs.nwjs.io/en/latest/

参考文

[1] JENSEN, P.B. (2017). 跨平台桌面应用程序：使用电子和 NW。 JS。曼宁出版公司。.

[2] KREDPATTANAKUL, K., and LIMPIYAKORN, Y. (2019). 使用 Electron 将基于 JavaScript 的 Web 应用程序转换为跨平台桌面。第九届国际信息科学与应用会议论文集（ICISA 2018）。 Springer, 新加坡, 第 571-579 页. DOI: 10.1007/978-981-13-1056-0.

[3] Benoit, A. (2015). NW.js 要点. Packt 出版有限公司.

[4] DIPIERRO, M. (2018). JavaScript 的崛起。计算机科学与工程, 20(1), pp. 9-10 页.

[5] PARSHINA, M. (2018). *JavaScript* 超越浏览器。信息技术学士学位论文。图尔库应用科技大学

[6] MARDAN, A. (2015). 节点简介。 JS。第 6 章在完整堆栈 JavaScript 中。Apress, Berkeley, CA, pp. 137-154 页 . DOI: 10.1007/978-1-4842-1751-1

[7] IHRIG, C.J. (2013). 节点模块系统。第 2 章在 Pro 节点中。 js 对于开发人员. Apress, Berkeley, CA, pp. 9-27 页.

[8] 电子文档。指南：Electron 入门。可从 https://electronjs.org/docs

[9] NW.js 0.13 及更高版本的文档。可从 http://docs.nwjs.io/en/latest/