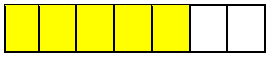
NodeJS/Electron

Version 1.0.0
Niveau requis : 4/7



Première application Standalone NodeJS packagé par Electron en exécutable Windows

Sommaire

I. Préambule

I.I. *Objet*

L'objet de ce document est de présenter la mise en place d'une application Windows construit par Electron.

I.II. *Prérequis*

Être sous Windows 10.

Avoir « git » installé sur son Poste.

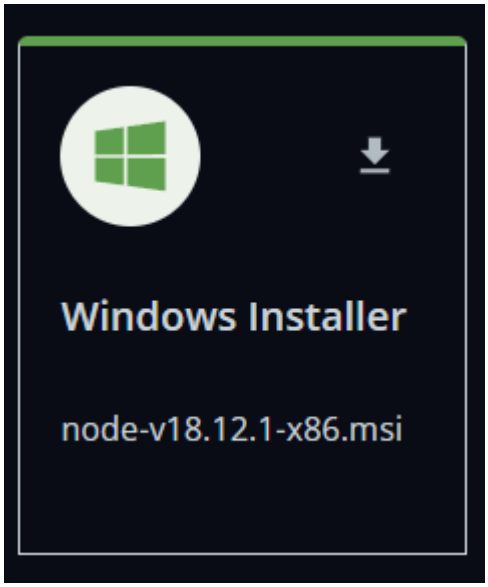
Partir d'une première installation de NodeJS avec NPM en bundle.

Si vous avez déjà une version de NodeJS avec NPM, veuillez la désinstaller en pensant préalablement vider le cache de npm :

```
npm cache clean –force
```

Supprimer les répertoires restants après désinstallation :
%USERPROFILE%\AppData\Roaming\npm
%USERPROFILE%\AppData\Roaming\npm-cache

Installer la version MSI de NodeJS : version actuelle LTS :



https://nodejs.org/fr/download/

I.III. *Versions du document*

Version	Date	Auteur	Description
1.0	30/12/2022	Péquignat.eu	Création du document

I.IV. *Documents de référence*

#	Document	Version	Auteur(s)
[R1]	https://nodejs.org/fr/download/	18.12.1	NodeJS
[R2]	https://www.electronjs.org/fr/docs/latest/tutorial/tutorial-prerequisites	N.A.	Electronjs tutorial
[R3]	https://www.electronforge.io/	N.A.	Electron Forge

II. Vérification de Node/NPM

Lancer dans un terminal CMD :

```
node -v
```

```
npm -v
```

```
D:\DEV\ELECTRON\my-electron-app>node -v
v18.12.1

D:\DEV\ELECTRON\my-electron-app>npm -v
8.19.2
```

III. Suivi du tutorial d'ElectronJS

III.I. *Développement d'une application Electron*

III.I.1 Création du projet NodeJS

Nous allons suivre le tutorial défini dans le document de référence [R2].

Nous allons par la suite utiliser Visual Studio Code de Microsoft qui est gratuit (VS Code).

Lancer dans le Terminal par exemple dans « D:\DEV\ELECTRON » les lignes de commandes :

```
mkdir my-electron-app
```

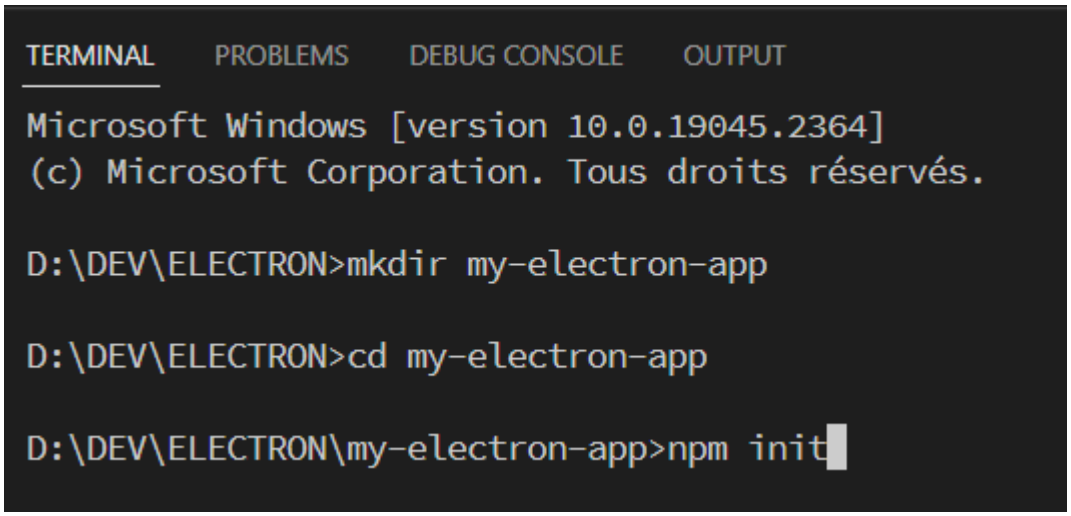
Créé un répertoire nommé « my-electron-app »
Puis

```
cd my-electron-app
```

Aller dans ce répertoire

```
npm init
```

Initialiser un projet NodeJS



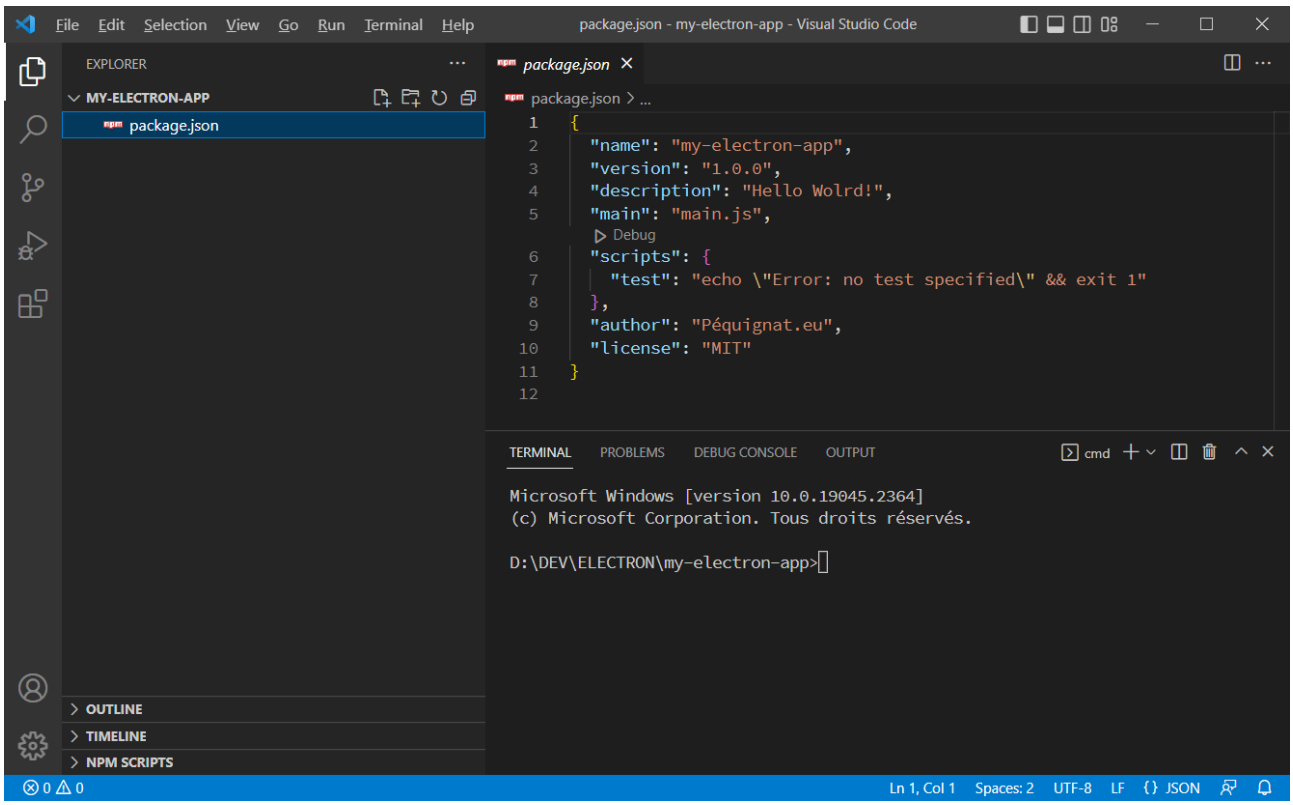
Répondez avec les éléments suivants :

Elément demandé	Réponse
package name:	my-electron-app
version: (1.0.0)	Laissez vide, Entrer
description:	Hello World !
entry point: (index.js)	Renseignez : « main.js »
test command:	Laissez vide, Entrer
git repository:	Laissez vide, Entrer
keywords:	Laissez vide, Entrer
author:	Ex : Péquignat.eu
license: (ISC)	Ex : MIT

```
{
  "name": "my-electron-app",
  "version": "1.0.0",
  "description": "Hello Wolrd!",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Péquignat.eu",
  "license": "MIT"
}


Is this OK? (yes)
```

Répondez par « yes »

Le fichier « package.json » est créé



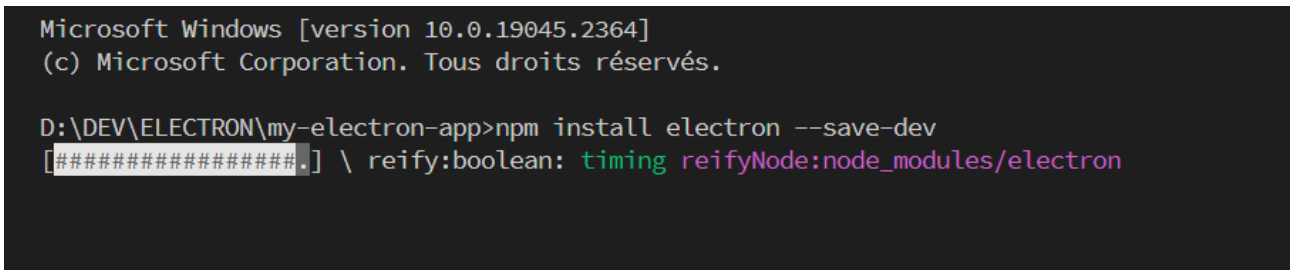
III.I.2 Installation de la dépendances Electron en tant que DEV

L'application Electron est utilisée lors de la phase de packaging en DEV pour construire un exécutable contenant sa propre instance d'exécutable de NodeJS.

C'est donc bien une dépendance de DEV qui n'est pas exploité en PRODUCTION.

L'installation passe en lançant la ligne de commande :

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



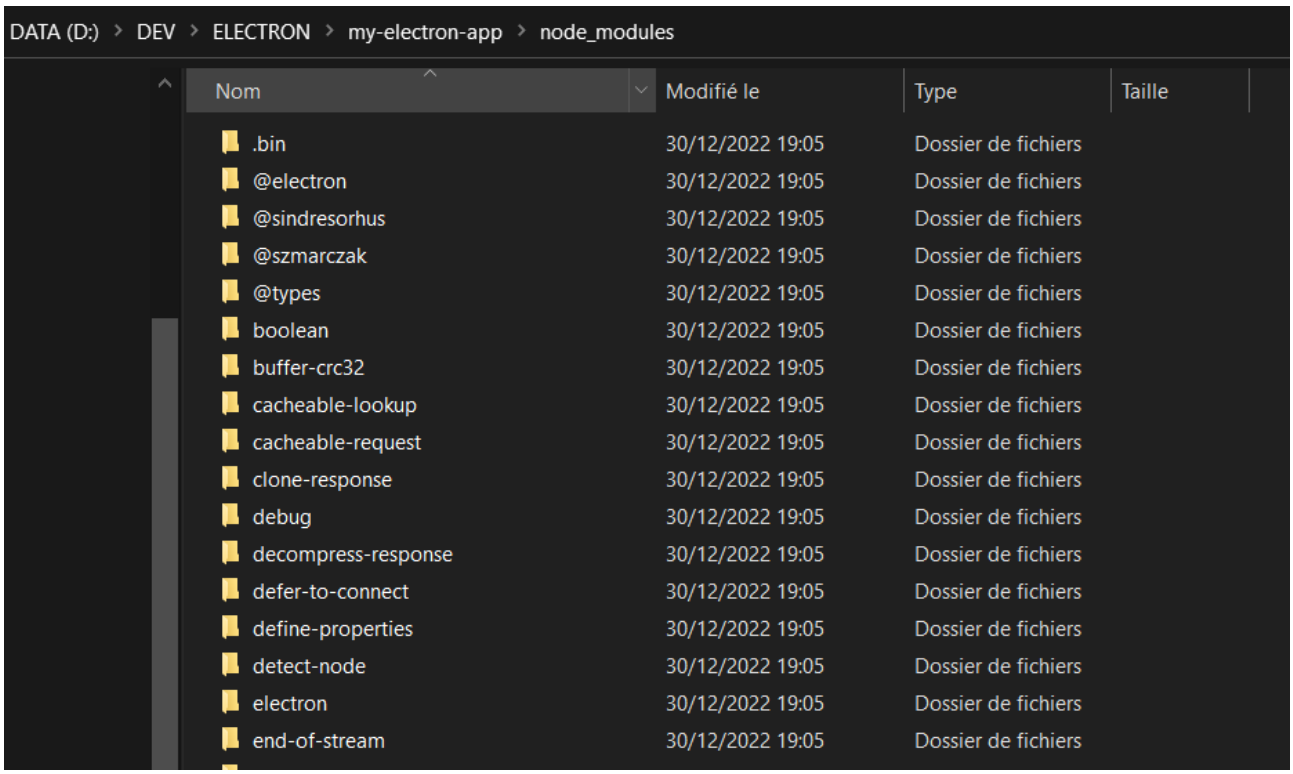
```
D:\DEV\ELECTRON\my-electron-app>npm install electron --save-dev

added 71 packages, and audited 72 packages in 16s

17 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

D:\DEV\ELECTRON\my-electron-app>
```

Vous pouvez vérifier que maintenant le fichier package.json contient la dépendances de DEV en plus :

```
{
  "name": "my-electron-app",
  "version": "1.0.0",
  "description": "Hello Wolrd!",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Péquignat.eu",
  "license": "MIT",
  "devDependencies": {
    "electron": "^22.0.0"
  }
}
```

Un nouveau fichier est présent « package-lock.json ».

De plus, l'Editeur masque le répertoire des dépendences « node_modules » :
« D:\DEV\ELECTRON\my-electron-app\node_modules »



III.I.3 Mise du fichier « .gitignore » et initialisation du repo Git

Afin d'éviter d'embarquer toutes les dépendances node_modules et autre fichier construit automatiquement, il est recommandé de créer un fichier « .gitignore » exemple :

https://github.com/github/gitignore/blob/main/Node.gitignore

```
# Logs
logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*
lerna-debug.log*
.pnpm-debug.log*

# Diagnostic reports (https://nodejs.org/api/report.html)
report.[0-9]*.[0-9]*.[0-9]*.[0-9]*.json

# Runtime data
```

```
pids
*.pid
*.seed
*.pid.lock

# Directory for instrumented libs generated by jscoverage/JSCover
lib-cov

# Coverage directory used by tools like istanbul
coverage
*.lcov

# nyc test coverage
.nyc_output

# Grunt intermediate storage (https://gruntjs.com/creating-plugins#storing-task-files)
.grunt

# Bower dependency directory (https://bower.io/)
bower_components

# node-waf configuration
.lock-wscript

# Compiled binary addons (https://nodejs.org/api/addons.html)
build/Release

# Dependency directories
node_modules/
jspm_packages/

# Snowpack dependency directory (https://snowpack.dev/)
web_modules/

# TypeScript cache
```

```
*.tsbuildinfo

# Optional npm cache directory
.npm

# Optional eslint cache
.eslintcache

# Optional stylelint cache
.stylelintcache

# Microbundle cache
.rpt2_cache/
.rts2_cache_cjs/
.rts2_cache_es/
.rts2_cache_umd/

# Optional REPL history
.node_repl_history

# Output of 'npm pack'
*.tgz

# Yarn Integrity file
.yarn-integrity

# dotenv environment variable files
.env
.env.development.local
.env.test.local
.env.production.local
.env.local

# parcel-bundler cache (https://parceljs.org/)
.cache
.parcel-cache
```

```
# Next.js build output
.next
out

# Nuxt.js build / generate output
.nuxt
dist

# Gatsby files
.cache/
# Comment in the public line in if your project uses Gatsby and not Next.js
# https://nextjs.org/blog/next-9-1#public-directory-support
# public

# vuepress build output
.vuepress/dist

# vuepress v2.x temp and cache directory
.temp
.cache

# Docusaurus cache and generated files
.docusaurus

# Serverless directories
.serverless/

# FuseBox cache
.fusebox/

# DynamoDB Local files
.dynamodb/

# TernJS port file
.tern-port
```

```
# Stores VSCode versions used for testing VSCode extensions
.vscode-test

# yarn v2
.yarn/cache
.yarn/unplugged
.yarn/build-state.yml
.yarn/install-state.gz
.pnp.*
```

Entrez la ligne de commande « git init »

```
D:\DEV\ELECTRON\my-electron-app>git init
Initialized empty Git repository in D:/DEV/ELECTRON/my-electron-app/.git/
```

Lancer la ligne de commande :

```
git add .gitignore package.json package-lock.json
```

```
D:\DEV\ELECTRON\my-electron-app>git add .gitignore package.json package-lock.json
warning: LF will be replaced by CRLF in package-lock.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in package.json.
The file will have its original line endings in your working directory.

D:\DEV\ELECTRON\my-electron-app>
```

Les fichiers ont été ajouté à git mais non comité.

III.I.4 Création du point d'entée main.js

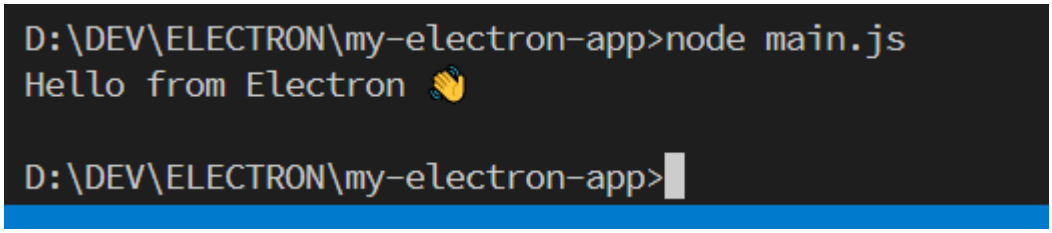
Créer un fichier à la racine du répertoire « my-electron-app » nommé « main.js »

Avec comme contenu :

```
console.log(`Hello from Electron 👋`)
```

Vérifiez son fonctionnement en lançant :

```
node main.js
```



Vous pouvez déjà ajouter ce fichier à Git :

```
git add main.js
```

III.I.5 Ajout du lancement par npm dans le package.json start

Dans le fichier « package.json », rajouter dans la partie « scripts » une première ligne

```
"start": "electron .",
```

Pour avoir le fichier de la forme :

package.json > ...

```
{
  "name": "my-electron-app",
  "version": "1.0.0",
  "description": "Hello Wolrd!",
  "main": "main.js",
  ▷ Debug
  "scripts": {
    "start": "electron .",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Péquignat.eu",
  "license": "MIT",
  "devDependencies": {
    "electron": "^22.0.0"
  }
}
```

Lancer ensuite en ligne de commande « npm run start »

```
D:\DEV\ELECTRON\my-electron-app>npm run start

> my-electron-app@1.0.0 start
> electron .


Hello from Electron  fæï
```

CTRL + C pour tuer le processus.

III.I.6 Création d'une page Web qui servira de vue Client de l'application electron

III.I.6.a Création du fichier index.html

Créer le fichier index.html à la racine du projet :

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8" />
  <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP -->
  <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'" />
  <meta http-equiv="X-Content-Security-Policy" content="default-src 'self'; script-src 'self'" />
  <title>Bonjour depuis le rendu d'Electron !</title>
</head>

<body>
  <h1>Bonjour depuis le rendu d'Electron !</h1>
  <p>👋</p>
</body>

</html>
```

Nous allons maintenant exploité electron comme conteneur cli pour afficher cette page web (Chromium).

Remplacer le contenu du fichier « main.js » par

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

const createWindow = () => {
  const win = new BrowserWindow({
```

```
    width: 800,
    height: 600,
  })

  win.loadFile('index.html')
}

app.whenReady().then(() => {
  createWindow()
})
```

III.I.6.b Explication

Ligne 1 : chargement des dépendances d'electron

- Le module **app**, qui contrôle le cycle de vie des événements de votre application.
- Le module **BrowserWindow**, qui crée et gère les fenêtres d'application.

Ligne 3 à 10 : Création d'une fonction qui sert à lancer la création de la fenêtre graphique qui charge le fichier index.html comme rendu.

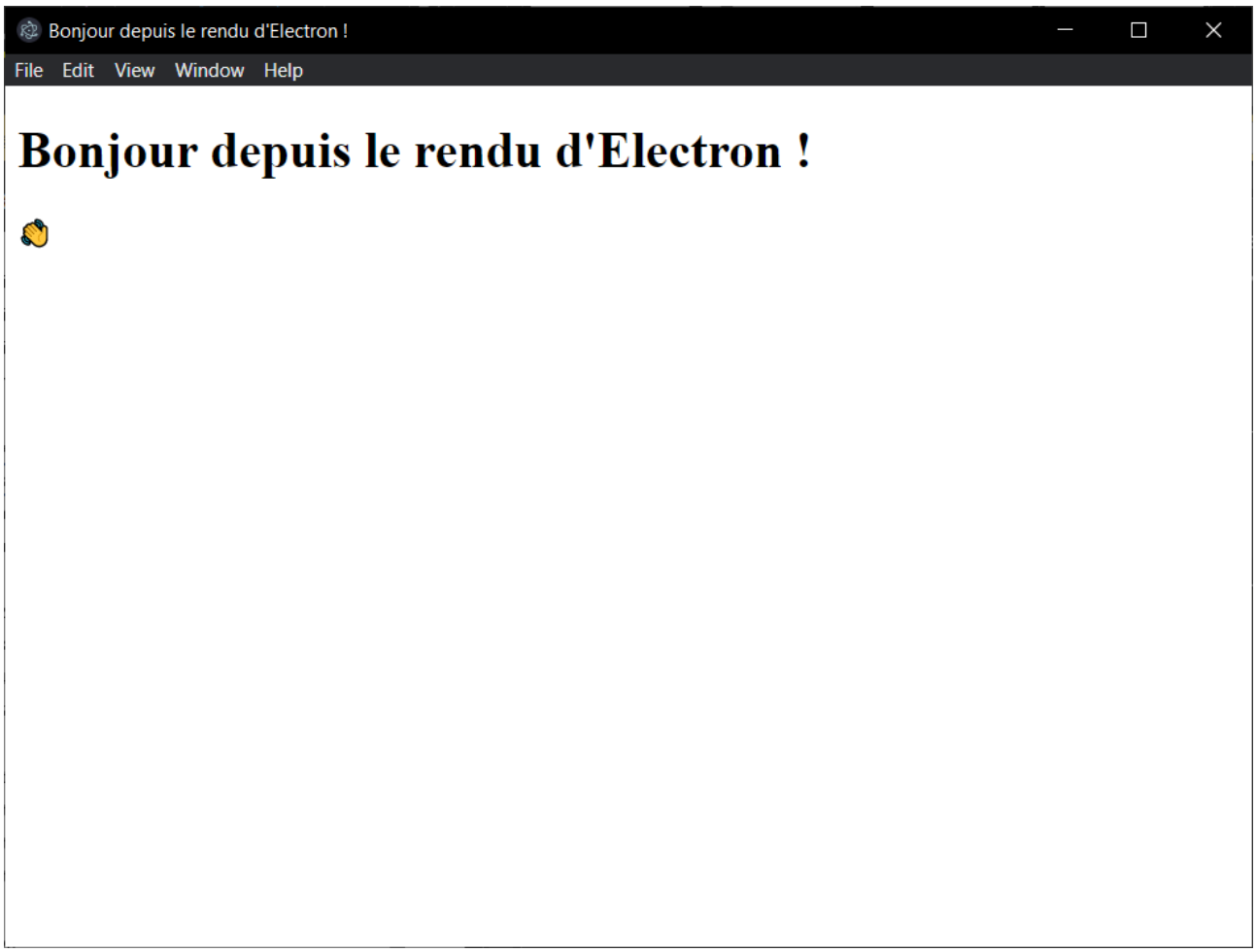
Ligne 12-14 : Déclenchement de l'ouverture de la fenêtre quand l'application est chargée et prête.

Note :
Dans Electron, BrowserWindows ne peut être créé qu'après le déclenchement de l'événement ready du module app. Vous pouvez attendre cet événement en utilisant l'API app.whenReady() et en appelant createWindow() une fois sa promesse réalisée.

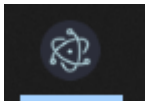
III.I.7 Exécution en DEV

Lancer dans le terminal : « npm run start »

La fenêtre Windows s'ouvre :



Dans la barre des tâches l'icône est celle d'électron !



Pour fermer l'application, utiliser la croix rouge en haut à droite de la fenêtre graphique.

III.I.8 Quitter l'application lorsque toutes les fenêtres sont fermées (Windows & Linux)

Sur Windows et Linux, le fait de fermer toutes les fenêtres ferme généralement entièrement l'application. Pour implémenter ce modèle dans votre application Electron, écoutez l'événement **window-all-closed** du module app, et appelez **app.quit**() pour quitter votre application si l'utilisateur n'est pas sur macOS.

```
app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') app.quit()
})
```

III.I.9 Ouverture d'une fenêtre si aucune n'est ouverte (macOS)

Par contre, les applications macOS continuent généralement à fonctionner même si aucune fenêtre n'est ouverte. L'activation de l'application lorsqu'aucune fenêtre n'est disponible va en ouvrir une nouvelle.

Pour implémenter cette fonctionnalité, écoutez l'événement activate du module app et appelez votre méthode createWindow() existante si aucune BrowserWindows n'est ouverte.

Étant donné que les fenêtres ne peuvent pas être créées avant l'événement ready , vous ne devrez écouter l'événement activate qu'après l'initialisation de votre application. Pour ce faire, écoutez uniquement les événements d'activation dans la callback de votre whenReady() existant.

```
app.whenReady().then(() => {
  createWindow()

  app.on('activate', () => {
    if (BrowserWindow.getAllWindows().length === 0) {
      createWindow();
    }
  });

})
```

III.II. *Debugger avec VS Code*

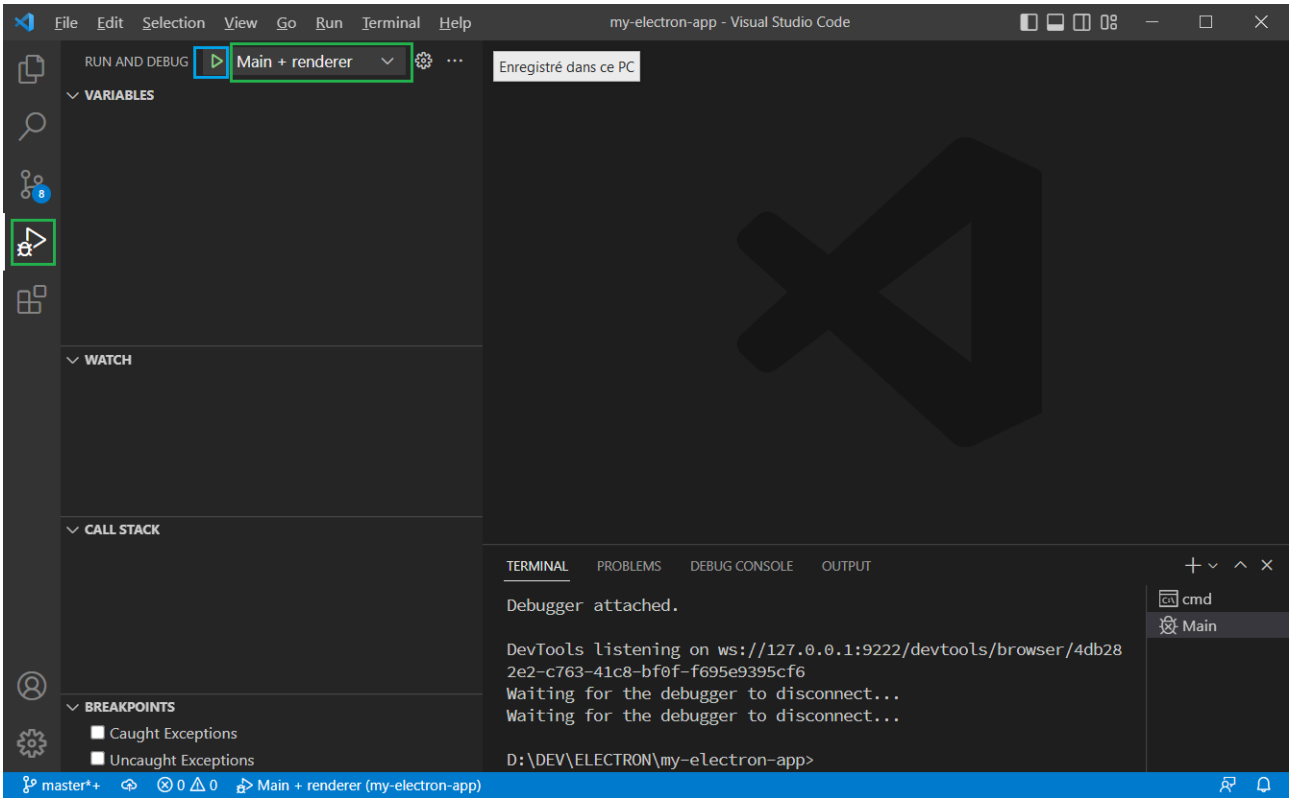
III.II.1 Création du fichier « .vscode/launch.json »

Créer le répertoire si non existant « .vscode » et dans ce répertoire, créer le fichier « launch.json » dont le contenu est le suivant :

```
{
  "version": "0.2.0",
```

```
  "compounds": [
    {
      "name": "Main + renderer",
      "configurations": [
        "Main",
        "Renderer"
      ],
      "stopAll": true
    }
  ],
  "configurations": [
    {
      "name": "Renderer",
      "port": 9222,
      "request": "attach",
      "type": "chrome",
      "webRoot": "${workspaceFolder}"
    },
    {
      "name": "Main",
      "type": "node",
      "request": "launch",
      "cwd": "${workspaceFolder}",
      "runtimeExecutable": "${workspaceFolder}/node_modules/.bin/electron",
      "windows": {
        "runtimeExecutable": "${workspaceFolder}/node_modules/.bin/electron.cmd"
      },
      "args": [
        ".",
        "--remote-debugging-port=9222"
      ],
      "outputCapture": "std",
      "console": "integratedTerminal"
    }
  ]
}
```

Fermer et relancer VS Code
Dans la partie Run de VS Code, sélectionner en haut « Main + renderer » :



Suite à cela la nouvelle option "Main + renderer" apparaîtra lorsque vous sélectionnez "Run and Debug" dans la barre latérale, en l'utilisant vous pourrez entre autres définir des points d'arrêt et inspecter toutes les variables dans les processus principal et de rendu.

Nous avonc dans ce fichier launch.json créé les 3 configurations suivantes:

- Main qui est utilisée pour démarrer le processus principal et expose également le port 9222 pour le débogage distant (--remote-debugging-port=9222). C'est le port que nous utiliserons pour attacher le débogueur au Renderer. Puisque le processus principal est un processus Node.js, le type est défini à node.
- Renderer, elle est utilisée pour déboguer le processus de rendu. Le processus principal étant celui qui crée ce processus, nous devons « nous y attacher » ("request": "attach") au lieu d'en créer un nouveau. Le processus de rendu est un processus web, donc le débogueur que nous devons utiliser est chrome.
- Main + renderer est une tâche composée qui exécute les deux précédentes et ceci simultanément.

III.III. *Utiliser des scripts de préchargement*

III.III.1 Objectif

Nous allons vouloir afficher la version de NodeJS embarqué, Chromium ainsi que de d'Electron.

III.III.2 Mise en œuvre

Pour cela, nous mettons à jour le fichier main.js avec un preload :

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

function createWindow () {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  })

  win.loadFile('index.html')
}

app.whenReady().then(() => {
  createWindow()

  app.on('activate', () => {
    if (BrowserWindow.getAllWindows().length === 0) {
      createWindow()
    }
  })
})
```

```
app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit()
  }
})
```

Le fichier preload.js correspondant :

```
window.addEventListener('DOMContentLoaded', () => {
  const replaceText = (selector, text) => {
    const element = document.getElementById(selector)
    if (element) element.innerText = text
  }

  for (const type of ['chrome', 'node', 'electron']) {
    replaceText(`${type}-version`, process.versions[type])
  }

  const loadingEl = document.getElementById("loading");
  loadingEl.setAttribute("style","display:none;");

  const infoPEl= document.getElementById("info");
  infoPEl.setAttribute("style","display:block");
```

Et le fichier index.html pour la partie présetation :

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <title>Hello World!</title>
  <meta http-equiv="Content-Security-Policy" content="script-src 'self' 'unsafe-inline';" />
</head>
```

```
<body>
  <h1>Hello World!</h1>
  <div id="loading" style="display:block;">Loading...</div>
  <p id="info" style="display:none;">
    We are using Node.js <span id="node-version"></span>,
    Chromium <span id="chrome-version"></span>,
    and Electron <span id="electron-version"></span>.
  </p>

</body>

</html>
```

III.IV. *Empaqueter votre application*

III.IV.1 Objectif

Dans l'optique de réaliser l'exécutable pour Windows, nous allons exploiter « Electron Forge ».

Electron n'inclut pas, dans ses modules de base, d'outil pour empaqueter et distribuer votre application. Une fois que votre application Electron est fonctionnelle en mode développement, vous allez devoir utiliser des outils supplémentaires pour créer une application packagée distribuable à vos utilisateurs (ce que l'on appelle communément un livrable). De tels livrables peuvent être soit des programmes d'installation (par exemple.MSI sous Windows) soit des fichiers exécutables portables (par exemple, .app sur macOS).

Electron Forge est un outil tout-en-un gérant l'empaquetage et la distribution des applications Electron. Sous le capot, il combine de nombreux outils Electron existants (par exemple, electron-packager, @electron/osx-sign, electron-winstaller, etc.) sous une seule interface afin que vous n'ayez pas à vous soucier de leurs interconnexions.

III.IV.2 Importation de votre projet dans Forge

Vous pouvez installer le CLI d'Electron Forge dans la section devDependencies de votre projet et importer votre projet existant avec un script de conversion très pratique.

```
npm install --save-dev @electron-forge/cli
```

```
D:\DEV\ELECTRON\my-electron-app>npm install --save-dev @electron-forge/cli
npm WARN deprecated @npmcli/move-file@2.0.1: This functionality has been moved to @npmcli/fs

added 345 packages, and audited 417 packages in 1m

62 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

```
npx electron-forge import
```

```
D:\DEV\ELECTRON\my-electron-app>npx electron-forge import
✓ Checking your system
✓ Locating importable project
✓ Processing configuration and dependencies
  ✓ Installing dependencies
  ✓ Copying base template Forge configuration
  ✓ Fixing .gitignore
✓ Finalizing import

› We have attempted to convert your app to be in a format that Electron Forge understands.

    …
```

Le fichier package.json est modifié pour donner :

```
{
  "name": "my-electron-app",
  "version": "1.0.0",
  "description": "Hello Wolrd!",
  "main": "main.js",
  "scripts": {
    "start": "electron-forge start",
    "test": "echo \"Error: no test specified\" && exit 1",
    "package": "electron-forge package",
    "make": "electron-forge make"
  },
  "author": "Péquignat.eu",
```

```
  "license": "MIT",
  "devDependencies": {
    "@electron-forge/cli": "^6.0.4",
    "@electron-forge/maker-deb": "^6.0.4",
    "@electron-forge/maker-rpm": "^6.0.4",
    "@electron-forge/maker-squirrel": "^6.0.4",
    "@electron-forge/maker-zip": "^6.0.4",
    "electron": "^22.0.0"
  },
  "dependencies": {
    "electron-squirrel-startup": "^1.0.0"
  }
}
```

En particulier la partie « scripts » qui exploite maintenant la forge et avec des commandes supplémentaires : « package » et « make »

Vous remarquerez également que votre fichier package.json a maintenant quelques paquets supplémentaires installés sous devDependencies et un nouveau fichier forge.config.js qui exporte un objet de configuration. Vous devriez voir plusieurs "makers" (packages qui génèrent des distribuables regroupés) dans la configuration préremplie, avec un pour chaque plate-forme cible.

III.IV.3 Création d'une distribution livrable

Pour créer un fichier distribuable, vous allez utiliser le nouveau script make de votre projet, qui exécutera la commande **electron-forge make**.

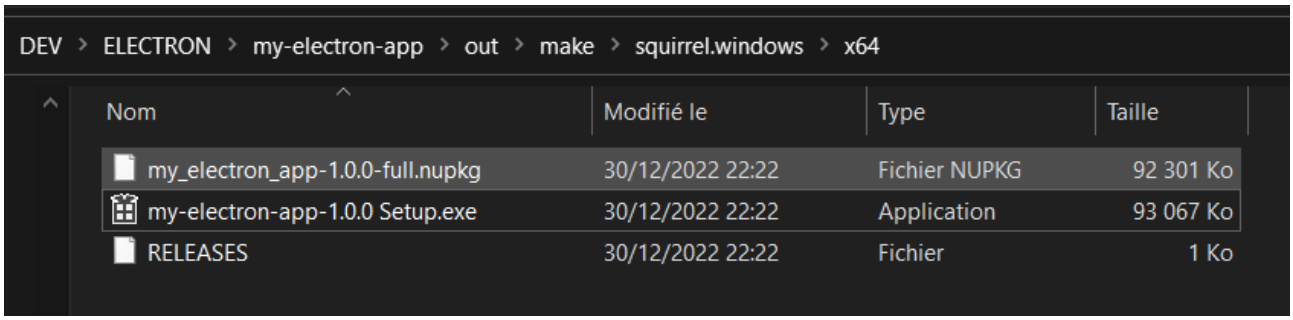
```
npm run make
```

```
D:\DEV\ELECTRON\my-electron-app>npm run make

> my-electron-app@1.0.0 make
> electron-forge make

✓ Checking your system
✓ Loading configuration
✓ Resolving make targets
  › Making for the following targets: squirrel
✓ Running package command
  ✓ Preparing to package application
  ✓ Running packaging hooks
    ✓ Running generateAssets hook
    ✓ Running prePackage hook
  ✓ Packaging application
    ✓ Packaging for x64 on win32 [7s]
  ✓ Running postPackage hook
✓ Running preMake hook
✓ Making distributables
  ✓ Making a squirrel distributable for win32/x64 [58s]
✓ Running postMake hook
  › Artifacts available at: D:\DEV\ELECTRON\my-electron-app\out\make
```

Le livrable se situe dans le répertoire « D:\DEV\ELECTRON\my-electron-app\out\make\squirrel.windows\x64 »

Toutefois dans le cas réel, le livrable a besoin d'être signé pour une vraie diffusion.

Note, une fois que les outils ont été installés, lors de la phase de make, aucun accès à internet n'est nécessaire dans cet exemple de tutorial.



IV.Fin du document