Prise en main d'Electron

1. Introduction à Electron

1.1.Pourquoi Electron ?

- Electron est un Framework de construction d'application multiplateformes fonctionnant sur Windows, MacOS et Linux.
- C'est un Framework utilisé pour créer des applications de bureau en utilisant des technologies Web (HTML/CSS/JS).
- Electron combine Chromium et Node.js dans son exécutable, avec la possibilité d'écrire du code natif personnalisé.
- Electron est une surcouche native pour les applications Web et est exécuté dans un environnement Node.js.
- Exemples connus : Visual Studio Code, Slack, Discord, Figma.

1.2.Avantages :

- Réutilisation des compétences Web (HTML/CSS/JS/TS/Vue).
- Déploiement multiplateforme (Windows/Mac/Linux).
- Accès aux API système (fichiers, notifications, menu, etc.).

1.3.Défis :

- Taille des exécutables (Electron embarque Chromium).
- Consommation de mémoire.
- Sécurité (importance du sandboxing et du contrôle des communications main/renderer).
    - **Le sandboxing** (ou exécution en bac à sable) dans une application Electron.js, consiste à exécuter les processus du rendu (renderer) dans un environnement isolé, similaire à celui du navigateur Chrome, pour renforcer la sécurité de l'application.

      Le rendu n'a pas accès directement à Node.js ou au système d'exploitation, ce qui limite les risques si du code malveillant s'exécute, car il ne pourra pas compromettre la sécurité de l'ordinateur.

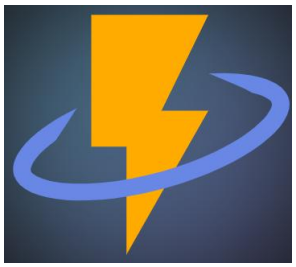
2. Stack choisie :

Electron

Node.js

TypeScript

Vue.js



Electron-Vite

Technologie	Rôle principal	Avantages clés
Electron	Conteneur desktop	- Accès natif aux APIs OS - Écosystème de plugins et extensions
Node.js	Exécution JavaScript côté « backend »	- Événementiel - Écosystème npm - Haute performance I/O
TypeScript	Typage statique et fonctionnalités avancées de JavaScript	- Pour grande application - Code plus structuré et moins de bugs de cohérence - Moins d'erreurs d'exécution - Transpile en JavaScript
Vue.js	Framework pour créer des interfaces utilisateur	- Réactivité avancée - Composants modulaires - Performances élevées
Electron-Vite	Bundler & dev-server pour Electron + Vue	- Bundler : outil qui regroupe plusieurs fichiers de code en un seul « paquet » optimisé pour être exécuté dans le navigateur ou dans une application (comme Electron). - Rechargement rapide HMR (Hot Module Replacement) ➔ Mise à jour instantanée sans redémarrer l'application - Builds optimisés

3. Environnement de développement

- Node.js et npm : outils fondamentaux pour le développement Web moderne et constituent le socle sur lequel repose les applications Electron.
- Node.js : un environnement d'exécution JavaScript basé sur le moteur JavaScript V8 de Chrome. Il permet d'exécuter du code JavaScript en dehors d'un navigateur Web.
- Electron exploite Node.js pour fournir un environnement d'exécution du code JavaScript.
- « npm » est le gestionnaire de packages Node. Il est distribué avec Node.js. Il simplifie la gestion et l'installation des différentes bibliothèques et dépendances nécessaires au projet Electron.

3.1. Installation de « Node.js » :

- Accéder au site Web de Node.js : https://nodejs.org/
- Télécharger l'installeur. Un fichier « .msi » pour Windows. Choisir la version appropriée : généralement deux options :
  - LTS (Long Term Support) : recommandée car elle est plus stable et bénéficie d'un support à long terme et de mise à jour de sécurité.
  - Version actuelle : cette version inclut les dernières fonctionnalités, mais peut être moins stable.
- Après l'installation, vérification avec les commandes :
  - `node -v` ou `node --version`
  - `npm -v`

NB : Si node.js ne fonctionne pas après l'installation, alors réinstaller Node via NVM pour Windows en ouvrant le terminal « cmd.exe » en tant qu'administrateur.

Exemple :

- `node -v  # pour savoir la version courante de node.js`

  v22.18.0
- `node -v 22.18.0`

  v22.18.0
- `nvm uninstall 22.18.0`
- `nvm install 22.18.0`
- `nvm use 22.18.0`
- `node -v`

3.2. Comprendre « npm » :

- Node Package Manager (npm), est le gestionnaire de paquets officiel de Node.js, utilisé pour installer, partager et gérer des bibliothèques JavaScript et pour automatiser les tâches.
- **Sert à** :
    - *Installer les modules* : librairies, outils et Frameworks dans le répertoire « **node_modules** » à la racine du projet.
    - *Gérer les dépendances* d'un projet via le fichier « **package.json** » : contient les métadonnées du projet, notamment son nom, sa version, ses dépendances et ses scripts.
    - *Exécuter les scripts :* automatiser les tâches (build, test, lancement de serveur, …).
- **Fichiers clés** :
    - « package.json » : décrit le projet, ses dépendances, ses scripts.
    - « node_modules » : dossier où npm installe les paquets.
    - « package-lock.json » : verrouille les versions exactes des dépendances.
- **Commandes « npm » courantes :**

Commande	Description
npm init	Initialise un projet Node.js et créé un fichier « package.json »
npm install <package>	Installe un paquet spécifique. Exemple : npm install electron
npm install	Installe toutes les dépendances du projet répertoriées dans le fichier « package.json ». Cette option peut être utilisée lors du clonage d'un projet à partir d'un dépôt.
npm update	Met à jour les dépendances
npm update <package>	Met à jour un paquet vers sa dernière version
npm uninstall <package>	Désinstalle un paquet spécifique. Exemple : npm uninstall electron
npm start	Exécute le script de démarrage défini dans le fichier « package.json » (dans « scripts », « start »).
npm run <script>	Exécute un script défini dans « package.json »

Liste complète des commandes : https://docs.npmjs.com/cli/v11/commands/npm

3.3. Éditeur recommandé : Visual Studio Code :

VS Code est aujourd'hui le standard pour le développement Web et Node.js pour différentes raisons :

- Open source et gratuit.
- Disponible sur toutes les plateformes (Windows, Linux, MacOs).
- Léger, rapide à installer et facile à configurer.
- Terminal intégré pour exécuter les scripts npm, les builds Vite, etc.

- Propose un écosystème extensible (extensions/plugins) pour tous les outils qui seront utilisés dans ce cours.
- Support fort de JavaScript/TypeScript : typage, autocomplétions, navigation de code, refactoring.

3.4.Installation de TypeScript :

Pour pouvoir exécuter des programmes TypeScript sur la machine, on pourrait installer globalement sur l'ordinateur par le biais de npm (Node Package Manager).

Vérifier si TypeScript est déjà installé sur l'ordinateur avec l'invite de commande (cmd.exe) ou via le terminal de VS Code :

```
tsc --version
```

- Commande d'installation :

```
npm install -g typescript
```

3.5.Installation et configuration du terminal « git bash » :

- Installation de « Git Bash » :

  Télécharger puis installer « Git pour Windows » à partir de son site officiel : https://git-scm.com/downloads/win

- Configuration de « Git Bash » comme *terminal par défaut* de « VS Code » :
  - Dans **VS Code**, Appuyer sur le raccourci « **Ctrl + Shift + P** » pour ouvrir la palette des commandes.
  - Taper « **Terminal: Select Default Profile** » et sélectionner cette option.
  - Dans la liste qui s'affiche, choisir « **Git Bash** ».

4. Premier exemple avec Electron

4.1.Initialisation du projet Electron

- Dans le terminal « Git Bash » de VS Code, créer le dossier « Exemple01 » pour le projet puis s'y positionner :

```
mkdir Exemple01
cd Exemple01
```

- La commande npm (Node Package Manager) provient du gestionnaire de paquets de Node.js, utilisé pour installer, gérer et partager des dépendances dans les projets JavaScript.

```
#initialisation du projet avec npm
npm init -y
```

➔ Permet de générer le fichier « package.json » contenant les informations du projet.

4.2.Installation d'Electron et de TypeScript dans le projet :

- Ajout d'Electron et de TypeScript avec leurs outils de développement :

```
npm install --save-dev electron typescript @types/node
```

Cette commande installe :

- electron : le runtime de l'application.
- typescript : le compilateur tsc
- @type/node : contient les types pour l'environnement Node.js (essentiel pour accéder à require, __dirname, etc.).

Cette commande génère le fichier « **package-lock.json** » et le dossier « **nodes_modules** » avec son contenu.

- Fichier « *package-lock.json* » : enregistre les versions exactes de toutes les dépendances (directes et indirectes) installées dans le projet. Garantit que chaque installation future utilise exactement les mêmes versions pour assurer la cohérence et éviter les bugs liés à des mises à jour imprévues.
- Dossier « *nodes_modules* » : contient toutes les dépendances (bibliothèques et modules) nécessaires au fonctionnement de l'application. Ces dépendances sont téléchargées depuis npm. Ce dossier permet au projet Electron d'accéder localement à ces modules sans avoir besoin de les retélécharger à chaque exécution de l'application.

4.3. Configuration de TypeScript :

- La configuration de compilation TypeScript pour le projet est assurée via le fichier « tsconfig.json ». Ce fichier contient par exemple la version de JavaScript cible et les répertoires d'entrée/sortie.
- Ce fichier permet de s'assurer que le code TypeScript de l'application est compilé de façon conforme aux options spécifiées.
- On peut générer le fichier « tsconfig.json » avec la commande :

```
npx tsc --init
```

- On va pour le moment utiliser la configuration suivante à la place de celle générée automatiquement dans le fichier :

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "CommonJS",
    "moduleResolution": "node",
    "esModuleInterop": true,
    "outDir": "./dist",
    "strict": true
  },
  "include": ["src/**/*"]
```

```
}
```

Les options du compilateur dans ce fichier sont :

- target : spécifie la version de JavaScript que TypeScript doit générer. Ici, TypeScript compilera le code en (ES2020).
- module : Spécifie le système de modules à utiliser lors de la compilation. CommonJS est le format utilisé par Node.js (avec require, module.exports).
- moduleResolution : indique comment localiser et résoudre les modules importés dans le projet (par exemple en suivant la logique de Node.js ou de TypeScript).
- esModuleIntrop : gérer l'importation des modules CommonJS en utilisant la syntaxe ES import, en évitant les erreurs lors de l'import de modules Node.js classiques.
- outDir : spécifie le dossier de sortie
- strict : Active toutes les vérifications strictes du compilateur TypeScript. Cela inclut notamment : noImplicitAny (évite les types implicites any), strictNullChecks (vérifie les null / undefined), strictFunctionTypes, etc.

L'option de configuration « include » : Indique à TypeScript quels fichiers compiler. Ici, tous les fichiers .ts (et .tsx) dans le dossier src et ses sous-dossiers sont inclus dans la compilation.

4.4. Création des fichiers de base

Créer le dossier « src » dans la racine du projet :

```
mkdir src
```

```
├── src/
│   ├── main.ts
│   ├── renderer.ts
├── index.html
├── package.json
└── tsconfig.json
```

- Création du fichier « index.html » dans la racine du projet :

```
touch index.html
```

Le fichier « index.html » contient la structure de base la page Web.

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Ma première app Electron</title>
</head>
<body>
    <h1>Bienvenue sur mon app Electron !</h1>
    <button id="bonjourBtn">Dire Bonjour</button>
```

```
    <script src="dist/renderer.js"></script>
</body>
</html>
```

- Création du fichier « main.ts » dans le dossier « src » pour le processus principal (Main Process) :

```
touch src/main.ts
```

Le fichier « main.ts » gère le processus principal (Main Process). Il démarre Electron et charge « index.html ».

```
import {app, BrowserWindow} from 'electron'

import path from 'path'

const createWindow = () => {
    const win = new BrowserWindow({
        width: 800,
        height: 600,

        webPreferences:{
            nodeIntegration: true
        }
    });

    win.loadFile(path.join(__dirname, '../index.html'));
};

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

NB :

« nodeIntegration » sert à autoriser ou interdire l'accès aux API Node.js dans les pages web (renderer) d'une application Electron. C'est un réglage très important, autant pour le fonctionnement que pour la sécurité.

Pour le moment, nous utilisons « nodeIntegration » avec la valeur « true ». Plus tard, nous verrons que c'est un peu risqué pour la sécurité de l'application. Nous mettrons cette valeur à **false** et nous accéderons aux API Node.js via un autre fichier « **preload.js** » (qui permet d'exposer des fonctions sécurisées au **renderer**).

- Création du fichier « src/renderer.ts » pour le processus de rendu (Renderer Process) :

Le Renderer Process contrôle l'interface utilisateur et affiche du contenu HTML/CSS/JS dans la fenêtre Electron.

```
touch src/renderer.ts
```

Permet d'interagir avec le DOM (HTML/CSS). On y met la logique côté « client » de l'application comme :

- Modification dynamique de l'interface.

- Réaction aux clics, formulaires, événements du DOM

- Communiquer avec le « main process » via « ipcRenderer »

Dans notre exemple actuel avec « **nodeIntegration : true** » dans « **src/main.ts** », le code du **Renderer** peut accéder directement aux « **APIs Node.js** ».

NB : On verra dans le chapitre suivant que cette approche est **dangereuse pour la sécurité**. On utilisera alors le script « **Preload** » combiné à « **l'IPC** » pour séparer les tâches entre processus de manière sécurisée.

```
console.log('Renderer est prêt');

document.addEventListener('DOMContentLoaded', () => {
    console.log('Electron is ready');
});


document.getElementById('bonjourBtn')?.addEventListener('click', () => {
    alert('Bonjour depuis le renderer process !');
    console.log('Bonjour depuis le renderer process !');
});
```

- Configuration du point d'exécution de l'application :

Ajout d'un script dans « package.json » pour qu'il soit possible de lancer l'application :

```
"main": "dist/main.js",

  "scripts": {
      "start": "electron .",
}
```

- Compilation des fichiers « .ts » :

La commande tsc (TypeScript Compiler) permet de transpiler des fichiers TypeScript (.ts) en JavaScript (.js).

Dans le terminal : `tsc`

L'exécution de la commande **tsc** permet de créer les fichiers « **main.js** » et « **renderer.js** » dans le dossier « **dist** » à partir des fichiers « main.ts » et « renderer.js » respectivement. Il faut vérifier leur création.

- Exécution de l'application Electron :

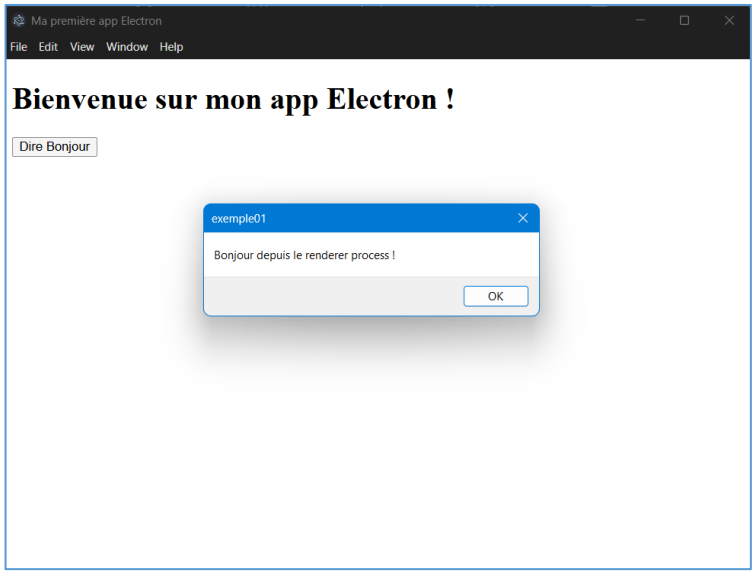
  Dans le terminal : `npm start`

Il est également possible d'utiliser la commande npx (un outil fourni avec npm à partir de la version 5.2) et permet d'exécuter directement un package Node.js sans avoir à l'installer globalement dans la machine.

Dans le terminal : `npx electron .`

ou

`npx electron dist/main.js`



4.5. Options de configuration de la fenêtre graphique Electron

La configuration de la fenêtre graphique (BrowserWindow) se fait lors de sa création, dans le fichier du Main Process « main.ts ». C'est là qu'on peut définir les propriétés d'apparence et de comportement de la fenêtre, via un objet d'options passé au constructeur de BrowserWindow.

- Options utiles de « BrowserWindow »

Option	Description
width, height	Dimensions initiales de la fenêtre Electron
minWidth, minHeight	Dimensions minimales de la fenêtre Electron
maxWidth, maxHeight	Dimensions maximales de la fenêtre Electron
resizable	Permettre ou non le redimensionnement

fullscreen	Ouvre la fenêtre Electron en plein écran
transparent	Rend la fenêtre Electron transparente
frame	Supprime le cadre natif (utile pour une interface personnalisée avec Vue/HTML)
autoHideMenuBar	Masque la barre de menu tant qu'on n'appuie pas sur Alt
backgroundColor	Définit la couleur d'arrière-plan
show	Indique si la fenêtre est visible au démarrage
webPreferences	Paramètres du contenu (sécurité, preload, etc.)
Icon	Change l'icône de la fenêtre

- Exemple :

Dupliquer le dossier « Exemple01 » et renommer-le « **Exemple02** » afin de tester différentes options de la fenêtre graphique d'Electron.

Dans le fichier « src/main.ts » :

Tester les options « BrowserWindow » présentées dans le tableau ci-dessus.

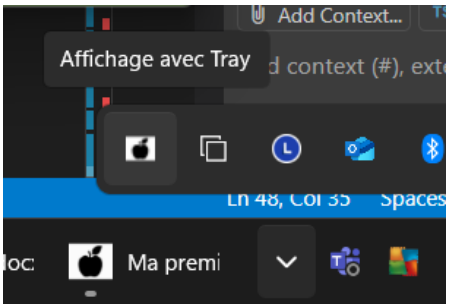
NB :

1- Pour l'option « backgroundColor », après avoir choisi la couleur dans la palette, il faut cliquer dans la zone en haut pour qu'elle soit appliquée.

2- Pour la modification de l'icône de la fenêtre de Electron, il faut que le fichier de l'image soit dans le dossier « dist » et qu'il soit un vrai fichier « .ico ». C'est-à-dire qu'il ne suffira pas de remplacer l'extension « .png » par « .ico ». Mais, il faut passer par un site de conversion vers « .ico ». Exemple : https://convertico.com/

- Affichage de l'icône dans la zone de notification :

Dans « src/main.ts », ajouter le code nécessaire pour afficher avec « Tray » l'icône de l'application dans la zone de notification.



4.6. Création d'un écran d'attente (fenêtre splash)

Une fenêtre splash (ou splash screen) dans une application Electron est une petite fenêtre qui s'affiche au lancement de l'application, avant que la fenêtre principale soit prête. C'est une sorte d'écran d'attente visuel.

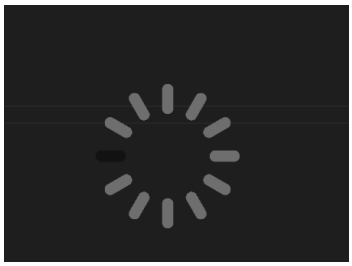
- Téléchargement d'une image animée « gif »
  - Pour télécharger une image « gif » animée : https://pixabay.com/fr/gifs
  - Télécharger une image de chargement et nommer-la « loading.gif »
  - La sauvegarder dans le dossier « dist/assets »
- Création de la page « splash.html » (au même niveau que « index.html ») :

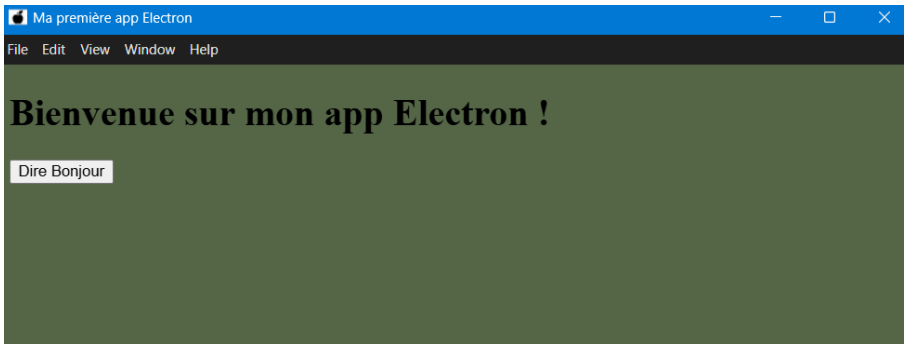
```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">

    <title>Fenêtre splash</title>
</head>
<body style="margin: 0; padding: 0; background-color: rgba(0, 0, 0, 0);">
    <img src="dist/assets/loading.gif" alt="Chargement..." width="150"
height="150"/>
</body>
</html>
```

- Fichier « src/main.ts » :

Ajouter le code permettant d'afficher un écran de chargement transparent, sans bordures et toujours en premier plan, à afficher pendant 5 secondes avant d'afficher la fenêtre principale.





4.7. Fenêtres parent et enfant, avec enfant une fenêtre modale

- Ajouts dans le fichier « src/main.ts » :

Ajouter le code permettant de définir puis afficher une nouvelle fenêtre modale après l'affichage de la fenêtre principale.