UTILISER ELECTRON JS COMME CONTENEUR D'APPLICATIONS HORS LIGNE

"Pouvoir utiliser des applications en mode hors ligne sur le terrain" est une demande forte des acteurs de l'humanitaire et du développement ayant accès à de faibles niveaux de connexion internet dans les contextes d'intervention. Peut se rajouter à cette demande, l'analyse de fichiers de données massifs techniquement complexes en hors ligne ou encore la nécessité d'avoir des applications facilement réutilisables pour accélérer leur création et leur implémentation. Autant de demandes et des contraintes que les équipes de CartONG s'efforcent de prendre en compte lorsqu'elles développent des applications web cartographiques en soutien des partenaires de notre structure.

Après une **courte présentation des solutions existantes** répondant aux contraintes du secteur, ce **document propose un tutoriel "étape par étape" de configuration d'Electron JS pour pouvoir l'utiliser comme conteneur d'applications hors ligne**. Vous trouverez aussi sur github l'archive contenant le container de démonstration. Bonne lecture !

I. Contexte

I.1. Conditions d'utilisation

Comme dans beaucoup d'autres secteurs où la cartographie est utilisée comme vecteur d'étude des données, le webmapping se développe depuis maintenant plusieurs années dans le secteur humanitaire. Offrant certains avantages comme la centralisation des données ou une possibilité accrue de diffusion et de communication, les applications web cartographiques peuvent également servir d'outils d'analyse et de reporting simplifiés (par opposition aux SIG bureautiques), ce qui les rend plus accessibles à des utilisateurs non spécialisés en cartographie. Par ailleurs, dans le contexte humanitaire, il est souvent nécessaire que ces applications web fonctionnent sans accès à internet.

Ce manque d'accès à internet sur le terrain est quasi-systématiquement associé à d'autres obligations. Les applications que nous développons pour nos partenaires doivent généralement :

- Pouvoir être utilisée en ligne / hors ligne : dans certains cas, l'utilisateur souhaite pouvoir diffuser sa version de l'application en l'hébergeant sur un serveur, dans d'autres cas l'utilisation se fera uniquement sur un ordinateur / serveur local. Le plus souvent, l'utilisateur l'utilise hors ligne sur le terrain puis la partage en ligne à son retour de mission.
- Fonctionner sans installation de logiciels tiers : les utilisateurs de nos applications n'ont généralement pas la possibilité d'installer des logiciels sur leur ordinateur (serveur web / SGBD / Docker / etc.) du fait des contraintes des droits d'administration.
- Permettre à l'utilisateur de mettre à jour lui-même les données utilisées par l'application : il doit avoir accès aux fichiers de l'application ou ajouter et mettre à jour ses données directement dans l'application.

- Avoir la capacité de gérer des jeux de données massifs : les fichiers Excel/CSV qui contiennent les données des applications peuvent contenir plusieurs dizaines de milliers de lignes et plusieurs dizaines de colonnes.
- Etre duplicable : un utilisateur a souvent besoin de dupliquer l'application pour avoir une version par région/thématique de travail avec des jeux de données différents.

I.2. Contraintes techniques

Contrairement aux sites web classiques qui sont hébergées sur un serveur physique et accessible via un serveur web, les sites web que nous fournissons à nos partenaires œuvrant sur le terrain doivent pouvoir être accessibles sans connexion à internet. Pendant des années, la solution la plus simple a été de construire des sites à partir de langages interprétables côté client et sans traitement côté serveurs, et de fournir à nos partenaire les fichiers sources des sites qu'ils pouvaient consulter directement en ouvrant le fichier index.html dans leur navigateur.

Comme dit ci-dessus, dans les applications web que nous produisons pour nos partenaires, il est le plus souvent nécessaire que les utilisateurs puissent modifier le jeu de données lu par l'application. S'affranchir des traitements côté serveur signifie l'impossibilité d'utiliser un SGBD pour stocker les jeux de données. Pour faire au plus simple pour les utilisateurs, les applications lisent des fichiers CSV/Excel/Géographiques directement présent dans le dossier de l'application que l'utilisateur récupère et ouvre avec son navigateur.

De nombreuses applications web développées par CartONG ont été réalisées durant une période où les navigateurs web préconisés pour nos utilisateurs (Google Chrome et Mozilla Firefox) autorisaient une page web à ouvrir automatiquement un fichier local (dans notre cas les jeux de données cartographiques). Pour des raisons de sécurité, les navigateurs bloquent désormais la lecture d'un fichier local par un fichier HTML local ouvert directement dans un navigateur : https://www.mozilla.org/en-US/security/advisories/mfsa2019-21/#CVE-2019-11730.

Cette restriction nous a obligé à repenser le protocole de diffusion de nos applications pour une utilisation hors ligne.

II. Étude des possibilités

Le tableau ci-dessous liste les possibilités envisageables pour répondre aux impératifs et problématiques précités :

Solution	Fonctionnement local	Sans installation	Accès aux fichiers par l'utilisateur	Gestion des jeux de données lourds	Duplicable	Pas besoin de recoder tout ou une partie de l'application
Fichiers (HTML/CSS/JS) copiés en local						
Serveur web local						
Serveur web local + SGBD						
Progressive Web App						
Empaquetage Electron						

***Figure 1 : Tableau comparatifs des solutions applicables pour une utilisation locale des applications web**. En rouge, les points bloquants. Les cases oranges indiquent que la solution n'est pas optimale pour ce besoin sans que cela ne soit bloquant. En vert, la solution est satisfaisante pour ce besoin.*

Les différentes restrictions techniques imposées (notamment l'impossibilité pour l'utilisateur d'installer des logiciels supplémentaires sur son poste de travail) limitent fortement les solutions applicables. Deux solutions permettent de répondre aux besoins des utilisateurs :

- La transformation des applications web classiques en Progressive Web Apps* (PWA). Cette technologie récente permet l'utilisation d'applications hors-ligne, mais elle nécessite des modifications sur les applications existantes pour que celles-ci soient fonctionnelles.
- Empaqueter les fichiers de code dans le générateur d'application Electron. Electron est un framework permettant de développer des applications multiplateformes de bureau avec des technologies web. L'infrastructure est basée sur Node.JS, et les applications sont affichées avec Chromium, le noyau open source de Google Chrome. Cette solution offre l'avantage de ne nécessiter aucun recodage des applications et d'être facilement duplicable.

Pour cette ressource et en lien avec le travail effectué, nous nous sommes tournés vers l'empaquetage des applications avec Electron. Le développement d'une PWA sera étudié plus en détail par CartONG dans le cadre du déploiement de futures applications.

III. Utilisation d'Electron comme conteneur d'applications hors-ligne

III.1. Processus general

Electron permet de construire des applications multi-plateformes (Windows / MacOS / Linux) en empaquetant des fichiers de code prévus pour un site web et en les transformant en un fichier exécutable comme un programme Desktop classique (Fig.2). Cette manœuvre a pour but d'éviter d'avoir à maintenir en parallèle plusieurs projets de développement dans le cas d'une application multi-plateforme.

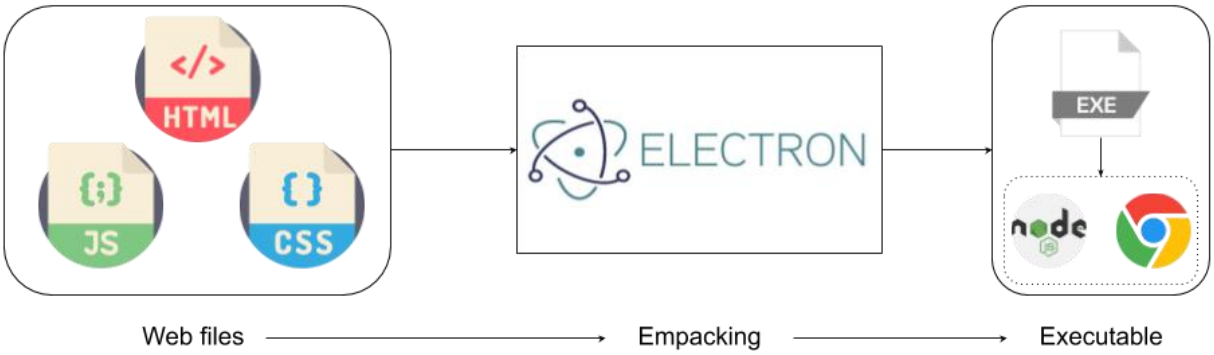


Figure 2 : Schéma du processus d'empaquetage

L'utilisation habituelle d'Electron diffère de l'objectif qui est le nôtre : à savoir empaqueter une application utilisable en hors ligne. Le processus d'empaquetage type qui produit un seul fichier exécutable en sortie n'est pas celui qui nous intéresse pour plusieurs raisons :

- Un fichier exécutable unique empêche l'utilisateur d'accéder aux fichiers de configuration et au jeu de données qui doit pouvoir être modifiable régulièrement. Ce problème pourrait être contourné en utilisant l'API de Node.JS qui est inclue dans Electron, mais cela demanderait un travail de recodage de l'application pour avoir accès aux modules de

Node dans les fichiers JS (c-à-d. le module "Path" qui permet d'interagir avec les fichiers systèmes).

- Le poids moyen constaté d'un fichier exécutable en sortie est de l'ordre de 40Mo, ce qui entraînerait une occupation conséquente de l'espace disque des utilisateurs en cas de multiplications des applications. C'est également un volume de données trop conséquent pour être diffusé dans le contexte d'une connexion internet à faible débit (ce qui est souvent le cas sur le terrain !).

III.2. Configuration d'un conteneur d'application

Basé sur Node.JS, la configuration d'Electron se fait via l'édition d'un fichier "package.json" et s'exécute dans une interface de ligne de commande (CLI) avec la commande "npm run". Pour l'empaquetage de l'application générée par Electron, nous avons utilisé "Electron-Forge" qui est un builder rapide et bien documenté.

L'exemple détaillé ci-dessous est basé sur l'application de démo fournie par Electron : il nécessite d'avoir Node.JS installé sur la machine de travail, d'avoir un minimum de connaissances sur cet outil et de savoir utiliser une console pour être bien compris.

1- Téléchargement de l'application de démo "Electron-Quick-Start"

```
git clone https://github.com/electron/electron-quick-start
```

A ce stade, si l'on observe le contenu du dossier téléchargé, on s'aperçoit que le fichier "package.json" est configuré de façon à ce que le fichier JS d'entrée soit le fichier "main.js" et que le script NPM "start" exécute Electron pour lancer l'application de démonstration :

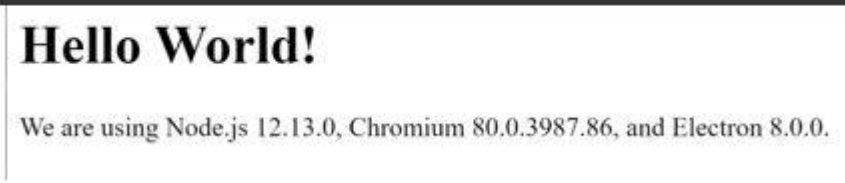
```
"main": "main.js",
  "scripts": {
      "start": "electron ."
  },
```

2- Avant d'exécuter le script "start", il est nécessaire d'installer les dépendances via NPM :

```
npm i
```

3- Une fois les dépendances installées nous pouvons lancer la commande pour ouvrir l'application de démo :

```
npm run start
```



4- Regardons maintenant le contenu du script main.js qui contient le code de configuration de l'application :

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

Electron fournit deux objets, “app” et “BrowserWindow” :

- L’objet “app” émets notamment les évènements qui se déclenchent au moment de l’ouverture/fermeture de l’application. Pour plus d’informations : https://www.electronjs.org/docs/api/app
- L’objet “BrowserWindow” crée et contrôle des fenêtres du navigateur chromium intégré dans Electron : https://www.electronjs.org/docs/api/browser-window

La gestion des évènements présents par défaut dans le script permet de gérer l’ouverture et la fermeture de l’application, le lancement de Chromium à l’ouverture de l’application s’effectue via l’événement “ready” :

```
app.on('ready', createWindow)
```

La fonction createWindow exécutée lorsque l’app émets l’évènement “ready” effectue plusieurs actions :

```
function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
      width: 800,
      height: 600,
      webPreferences: {
      preload: path.join(__dirname, 'preload.js')
      }
  })
  // and load the index.html of the app.
  mainWindow.loadFile('index.html')

  // Open the DevTools.
  mainWindow.webContents.openDevTools()
}
```

C’est dans cette fonction qu’est créée la fenêtre qui affichera les fichiers web de notre application via l’objet “BrowserWindow” et la méthode “BrowserWindow.loadFile()”. Ici c’est le fichier index.html qui est à la racine du dossier qui est chargé.

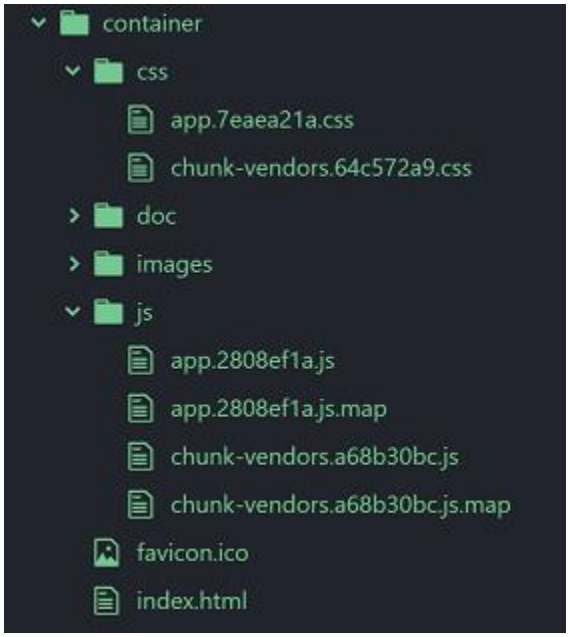
L’objet “BrowserWindow” contient également une propriété “webPreferences” de type objet donc la propriété “preload” permet d’exécuter un script avant les autres scripts chargés dans la page. Cette option de pré-chargement de script est intéressante car elle a accès à tous les modules Node (comme le module Path cité plus haut).

“BrowserWindow” offre également la possibilité d’ouvrir par défaut les outils de développements de Chromium via la propriété webContents.openDevTools().

5- Nous allons modifier ce fichier pour obtenir le code suivant :

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

function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
      width: 800,
      height: 600,
  })
  mainWindow.maximize();
 mainWindow.loadURL(`file://${__dirname}/container/index.html`);
}

app.on('ready', createWindow)

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

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

Nous conservons ainsi les fonctions d'ouverture/fermeture de l'application par défaut, mais nous enlevons le chargement d'un fichier avec la méthode "preload", nous utilisons la méthode "maximize()" pour passer l'application en plein écran, et nous changeons le chemin du fichier HTML à charger pour que celui soit situé dans un dossier nommé "container". Nous plaçons ensuite les fichiers de l'application web que nous voulons empaqueter dans Electron dans ce dossier "container" pour avoir l'arborescence suivante :



Si on relance la commande “npm run start”, notre application s’affiche correctement.

6- Il faut à présent empaqueter l’application pour qu’elle puisse être utilisée sans Node. JS.Electron ne propose pas directement de méthode d’empaquetage complète, mais il existe plusieurs packages NPM pour cette opération. Nous allons ici utiliser le package electron-forge : https://github.com/electron-userland/electron-forge

La première étape consiste à ajouter la dépendance de développement electron-builder à notre projet :

```
npm i -D electron-forge
```

Nous allons également ajouter le package “maker-squirrel” qui va être utilisé pour empaqueter une application pour une utilisation sous Windows.

```
npm i @electron-forge/maker-squirrel
npm i electron-squirrel-startup
```

7- Nous allons ensuite modifier le fichier “package.json” pour obtenir le code suivant :

```
{
  "name": "CartONG-App-Container",
  "productName": "CartONG-App-Container",
  "description": "Electron container for web applications",
  "version": "1.0.0",
  "main": "src/index.js",
  "scripts": {
      "start": "electron-forge start",
      "package": "electron-forge package",
      "make": "electron-forge make"
  },
  "keywords": [],
  "author": {
      "name": "Olivier Ribiere",
      "email": "o_ribiere@cartong.org"
  },
  "license": "MIT",
  "config": {
      "forge": {
      "packagerConfig": {
      "icon": "favicon.ico"
      },
      "makers": [
      {
            "name": "@electron-forge/maker-squirrel",
            "config": {
            "name": "epimap"
            }
      }
      ]
      }
  },
  "dependencies": {
      "@electron-forge/maker-squirrel": "^6.0.0-beta.45",
      "electron-squirrel-startup": "^1.0.0"
  },
  "devDependencies": {
      "@electron-forge/cli": "^6.0.0-beta.45",
      "@electron-forge/maker-zip": "^6.0.0-beta.45",
      "electron": "6.0.10"
  }
}
```

Electron-Forge offre une série de makers en fonction de la plateforme sur laquelle l'application doit servir (Linux/MacOs/Windows/etc). Dans notre cas, nos applications doivent tourner sous Windows.

Nous avons ajouté la paire clé/valeur suivante à l'objet "script" :

```
"make": "electron-forge make"
```

Ce script lance le maker "maker-squirrel" :
https://www.electronforge.io/config/makers/squirrel.windows

"The Squirrel.Windows target builds a number of files required to distribute apps using the Squirrel.Windows framework. It generates a {appName} Setup.exe file which is the main installer for your application."

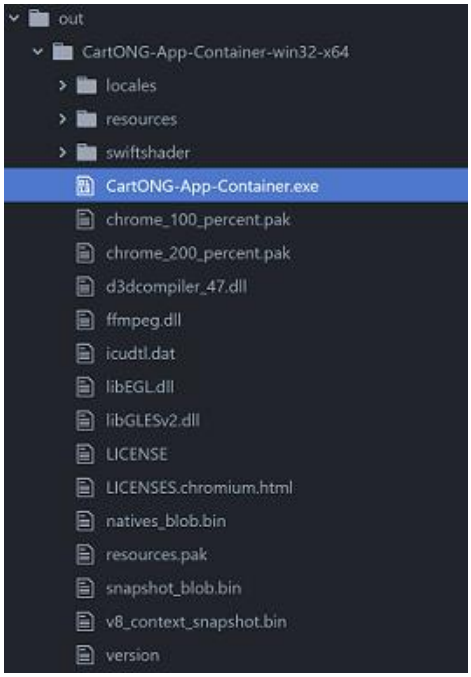
Cette étape revient à créer un fichier exécutable non empaqueté, ce qui est indispensable pour que l'utilisateur puisse avoir accès aux fichiers de l'application.

Nous avons également spécifié les propriétés de la clé "forge" qui contient la configuration d'electron-forge :

```
"forge": {
      "makers": [
      {
            "name": "@electron-forge/maker-squirrel"
      }
      ]
}
```

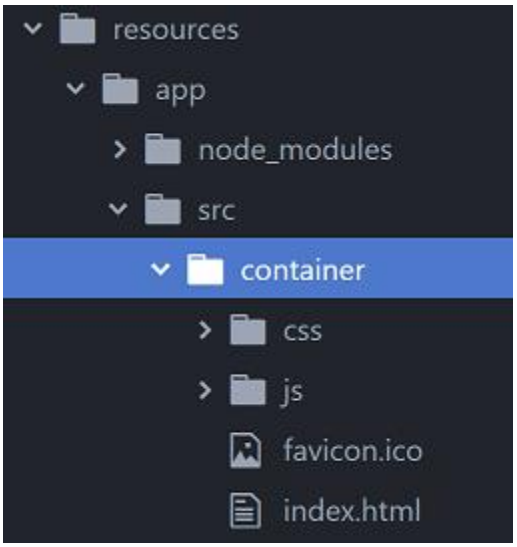
La clé "makers" contient la liste des makers à utiliser, dans notre cas, uniquement "maker-squirrel" est utilisé, ce maker ne contient pas d'options de configuration.

8- En lançant la commande "npm run make", un dossier "out" est créé avec l'arborescence suivante :



A la racine du dossier, se trouve le fichier "CartONG-App-Container.exe". Si on exécute ce fichier, notre application s'ouvre correctement.

Le dossier "resources" contient les fichiers de l'application dans le répertoire "app/src/container/" :



Ces fichiers peuvent être remplacés par les fichiers d'une autre application web, Electron se contente de chercher la page index.html dans ce dossier et de l'ouvrir dans Chromium.

En sortie, nous obtenons une archive de 74 Mo. Ce poids n'est pas négligeable, mais cette archive a l'avantage de ne devoir être récupérée qu'une seule fois, seuls les fichiers du site web seront à mettre à jour ensuite.

L'utilisateur n'a ainsi plus qu'à télécharger les fichiers webs qu'il désire, les insérer dans le dossier "container" et lancer l'exécutable pour profiter de son application web sans avoir besoin d'installer un serveur web en local (Fig.4), et sans qu'aucun re-développement de l'application originale ne soit nécessaire. Pour les utilisateurs souhaitant se partager des applications web dans un contexte de débit restreint, il leur est ainsi possible de n'envoyer que les fichiers du site à un autre utilisateur, qui pourra ensuite également les consulter sur son container Electron.

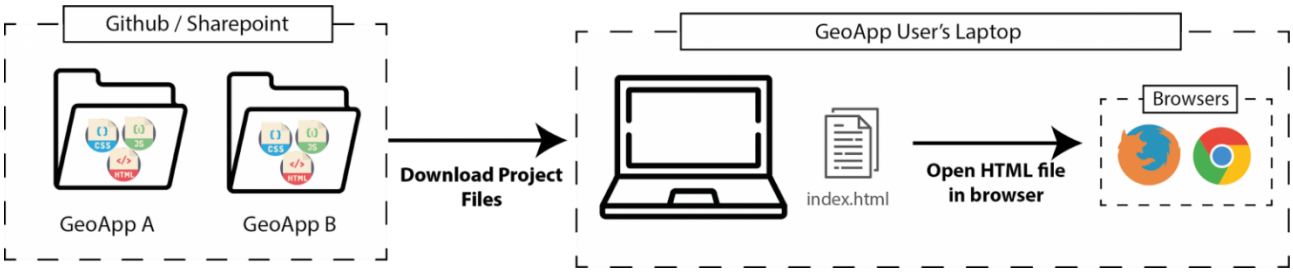


Figure 3 : workflow de départ

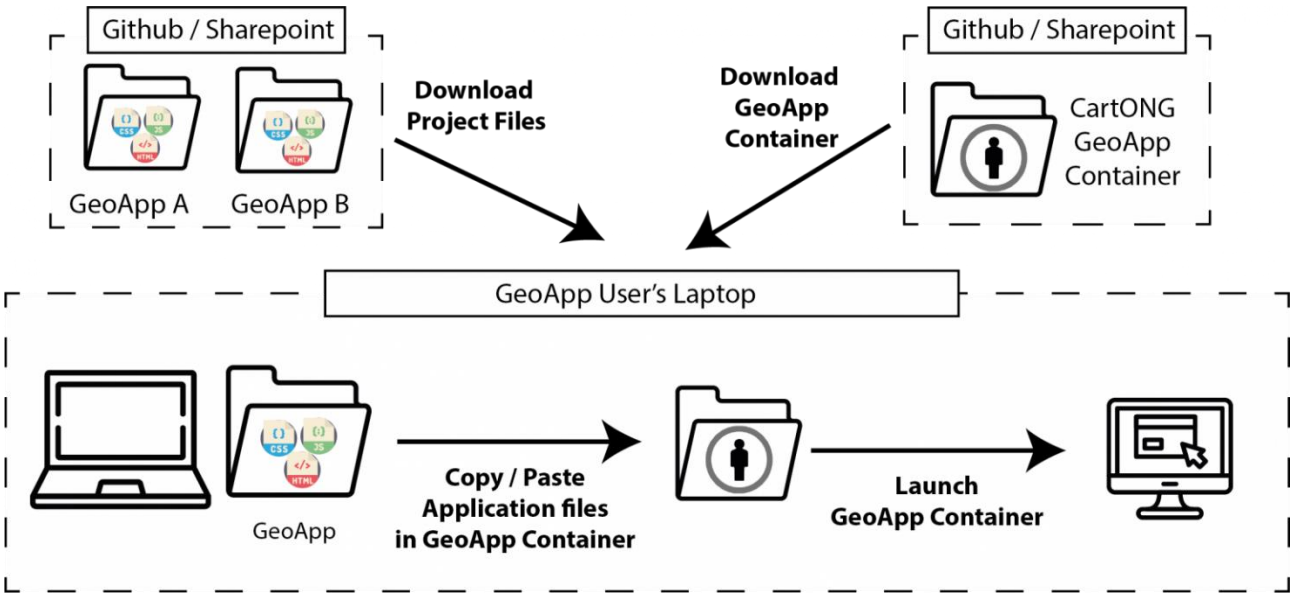


Figure 4 : Nouveau workflow

L'archive contenant le container de démonstration est disponible sur github : https://github.com/CartONG/Electron-App-Container