Auteur externe

Zeste de savoir

Vos applications avec Electron

12 août 2019

Table des matières

Vous aimeriez créer une application pour bureau, mais vos compétences se limitent aux technologies du Web comme HTML5, CSS3 et JavaScript ? Ça tombe bien, Electron est là !

Electron vous permettra de créer des applications parfaitement intégrées à Windows, Linux ou Mac OS X simplement avec du HTML, du CSS et un peu de JavaScript.

De nombreuses applications comme Atom, Visual Studio Code, ou encore le navigateur Brave sont basées sur Electron et permettent de créer rapidement de belles interfaces parfaitement intégrées au système d'exploitation, quel qu'il soit.

Prérequis nécessaires
Connaître un minimum HTML5, CSS3 et JavaScript.
Ne pas avoir peur d'utiliser la ligne de commande et savoir l'utiliser basiquement (ouverture, `cd` …).
Disposer de Node.js, ou au moins de npm.
Prérequis optionnels

Connaître les APIs de Node.js.
Connaître JavaScript dans sa version ES6.
Objectifs
Découvrir Electron, ainsi que certaines APIs qui lui sont spécifiques.
Savoir utiliser `electron-packager` pour distribuer ces applications.

1. Installer Electron

Electron est disponible sous la forme d'un package npm. L'installer sera donc très simple, et tiendra en une ligne.

Pour commencer, ouvrez un terminal en mode *Administrateur* ou en *root* (selon les systèmes), puis exécutez ceci.

```
1 npm install electron -g
```

Le package 'electron est une version d'Electron prête à l'emploi. Une fois que c'est terminé, vous pouvez vérifier que tout s'est bien installé en tapant cette commande.

```
1 electron --version
```

Ce qui devrait simplement vous afficher la version dont vous disposez. Et voilà, l'installation est déjà terminée, ce n'était pas très compliqué.

Dans ce tutoriel, j'utilise la version 1.4.1 d'Electron.

2. Créer une première application

Comme vous allez le voir, le fait qu'Electron repose sur les technologies du Web vous permet d'avoir des applications fonctionnelles en quelques lignes de code seulement. Pour un *Hello world*, nous n'aurons besoin que de trois fichiers :

— Le manifeste de l'application ;
— Le script principal, qui lance les différentes fenêtres ;
— Une page HTML à afficher

2.1. Créer le manifeste

Le manifeste d'une application Electron est semblable à celui de n'importe quel site ou application Node.js : il est écrit dans un fichier `package.json`, et contient les mêmes clés.

Les trois clés dont nous aurons besoin au minimum sont les suivantes :

— `name`, le nom de notre application ;
— `version`, la version de l'application ;
— `main`, le chemin vers le script principal.

Nous allons donc créer notre manifeste grâce à la commande `npm init`. Au final vous devriez avoir quelque chose qui ressemble à ceci.

```
{
  "name": "tutoriel-electron",
  "version": "1.0.0",
  "description":
      "Une application pour explorer les possibilités d'Electron",
  "main": "main.js",
  "scripts": {
    "test": "electron ."
  },
  "author": "Bat'",
  "license": "GPL-3.0"
}
```

Je vous conseille aussi d'exécuter cette commande, après vous être placé dans le répertoire de votre nouveau projet.

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

Chez vous, il ne se passera rien, mais la dépendance à Electron sera ajoutée dans le manifeste de votre projet, et si quelqu'un veut vous aider en sans avoir installé Electron avant, il sera installé automatiquement après un `npm install`.

2.2. Le script principal

Maintenant que notre manifeste est prêt, on va devoir créer notre script principal. Ce script est un fichier JavaScript qui va initialiser les différentes fenêtres de notre application, mais c'est aussi le seul à pouvoir accéder à certaines APIs, comme celle pour ouvrir des boîtes de dialogue (mais on verra ensuite qu'il y a des moyens de contourner cette limitation).

Dans ce tutoriel, j'utiliserai des fonctions d'ECMAScript 6, comme les mot-clés `let` et `const` ou encore les fonctions fléchées, puisqu'elles sont supportées par Electron. Si vous ne savez comment les utiliser, vous pouvez regarder respectivement cet article et celui-ci .

Nous allons commencer par importer le seul module dont nous aurons besoin, `electron` bien-sûr. Pour plus de simplicité nous allons aussi définir de petits raccourcis vers des objets que nous allons utiliser ensuite.

```
1 const electron = require('electron');
2 const app = electron.app;
3 const BrowserWindow = electron.BrowserWindow;
```

À la suite de ceci, on va ajouter ce code qui initialise une fenêtre.

```
1  let mainWindow;
2
3  function createWindow () {
4
5    mainWindow = new BrowserWindow({width: 1800, height: 1200}); //
         on définit une taille pour notre fenêtre
6
7    mainWindow.loadURL(`file://${__dirname}/index.html`); // on doit
         charger un chemin absolu
8
9    mainWindow.on('closed', () => {
10     mainWindow = null;
11   });
12 }
```

Ici nous créons une variable globale, `mainWindow`, qui contiendra notre fenêtre principale. Après cela, nous allons créer une fonction nommée `createWindow` qui initialisera notre cette fenêtre. Dans cette fonction, nous allons tout d'abord créer une nouvelle instance de la classe `BrowserWindow`, qui représente une fenêtre. Elle s'affichera automatiquement. Ensuite, nous allons y charger notre page HTML, stockée dans le fichier `index.html`. Enfin, lorsque la fenêtre est fermée, on la définit à `null`, vous verrez que ça nous sera utile pour la suite. Pour cela, il faut utiliser l'évènement `closed` de la fenêtre principale.

Comme vous pouvez le remarquer, les objets d'Electron ont une méthode `on` qui permet d'écouter leurs évènements. Ici, à la ligne 9, on écoute l'évènement `closed` de la fenêtre qui correspond au moment où elle est fermée.

Maintenant, il faut appeler la méthode que nous venons de créer lorsque l'application démarre. Pour cela, il faut la lier avec l'évènement `ready` de notre constante `app`. Ainsi on est sûr qu'Electron a bien fini de s'initialiser et qu'on peut utiliser sans problèmes ses APIs.

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

Mais le script principal continuera à tourner même quand on aura fermée notre fenêtre. On va devoir utiliser l'évènement `window-all-closed` d'`app`, qui est émit quand toutes les fenêtres ont étés fermées. À ce moment, on va utiliser la méthode `app.quit` qui va nous permettre de quitter l'application.

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

Comme vous pouvez le remarquer, j'utilise une condition qui vérifie qu'on est pas sous Mac OS X avant de fermer notre application. En effet, en général les applications sous cette plateforme ne se ferment pas mais continuent de tourner en arrière-plan. Cependant, ce comportement n'est pas du tout obligatoire.

Mais comme on a implémenté ce comportement on va avoir un problème. Si un utilisateur qui a un Mac lance l'application pour la première fois en cliquant sur l'icône dans le Dock, il verra bien notre fenêtre s'afficher. Maintenant s'il la ferme, l'application ne s'arrêtera pas. Et lorsqu'il cliquera de nouveau sur l'icône, il ne verra rien, car l'évènement `ready` d'`app` ne sera pas émit, puisqu'il ne l'est qu'une fois, au démarrage de notre application.

Pour contrer ce problème, on va utiliser l'évènement `activate` de la constante `app`, qui lui est déclenché quand on clique sur l'icône dans le Dock.

```
app.on('activate', () => {
  if (mainWindow === null) {
    createWindow();
  }
});
```

Ici, vous remarquez qu'on vérifie d'abord que la fenêtre n'existe pas encore (c'est là que donner la valeur de `null` à la fermeture de la fenêtre est utile), puis si c'est le cas, on la crée.

Finalement, notre fichier ressemble à ceci :

```
const electron = require('electron');
const app = electron.app;
const BrowserWindow = electron.BrowserWindow;

let mainWindow;
```

```
6
7  function createWindow () {
8
9    mainWindow = new BrowserWindow({width: 1800, height: 1200});
10
11   mainWindow.loadURL(`file://${__dirname}/index.html`);
12
13   mainWindow.on('closed', () => {
14     mainWindow = null;
15   })
16 }
17
18 app.on('ready', createWindow);
19
20 app.on('window-all-closed', () => {
21   if (process.platform !== 'darwin') {
22     app.quit();
23   }
24 });
25
26 app.on('activate', () => {
27   if (mainWindow === null) {
28     createWindow();
29   }
30 });
```

Vous avez ici le code minimal pour à peu près n'importe quelle application Electron. Il ne nous reste plus qu'une chose à faire, créer notre page HTML !

2.3. L'interface : un peu de HTML

Dans notre script principal, nous chargeons une page appelée `index.html`, on va donc écrire notre HTML dans ce fichier. On va faire quelque chose de très simple : du HTML minimal, avec un titre (`h1`), « Hello world ! ». Mais si vous voulez, vous pouvez faire plus : ajouter du texte, des images, du CSS …

Maintenant que nous avons créé notre interface, nous n'avons plus qu'à lancer notre application. Pour cela, ouvrez un terminal dans le dossier de votre code, puis tapez cette commande.

```
1 electron .
```

Ici, on lance Electron, en lui donnant comme argument le dossier actuel, celui de notre code. Il va alors aller lire notre manifeste, où il trouvera le chemin vers le script principal et il l'exécutera.

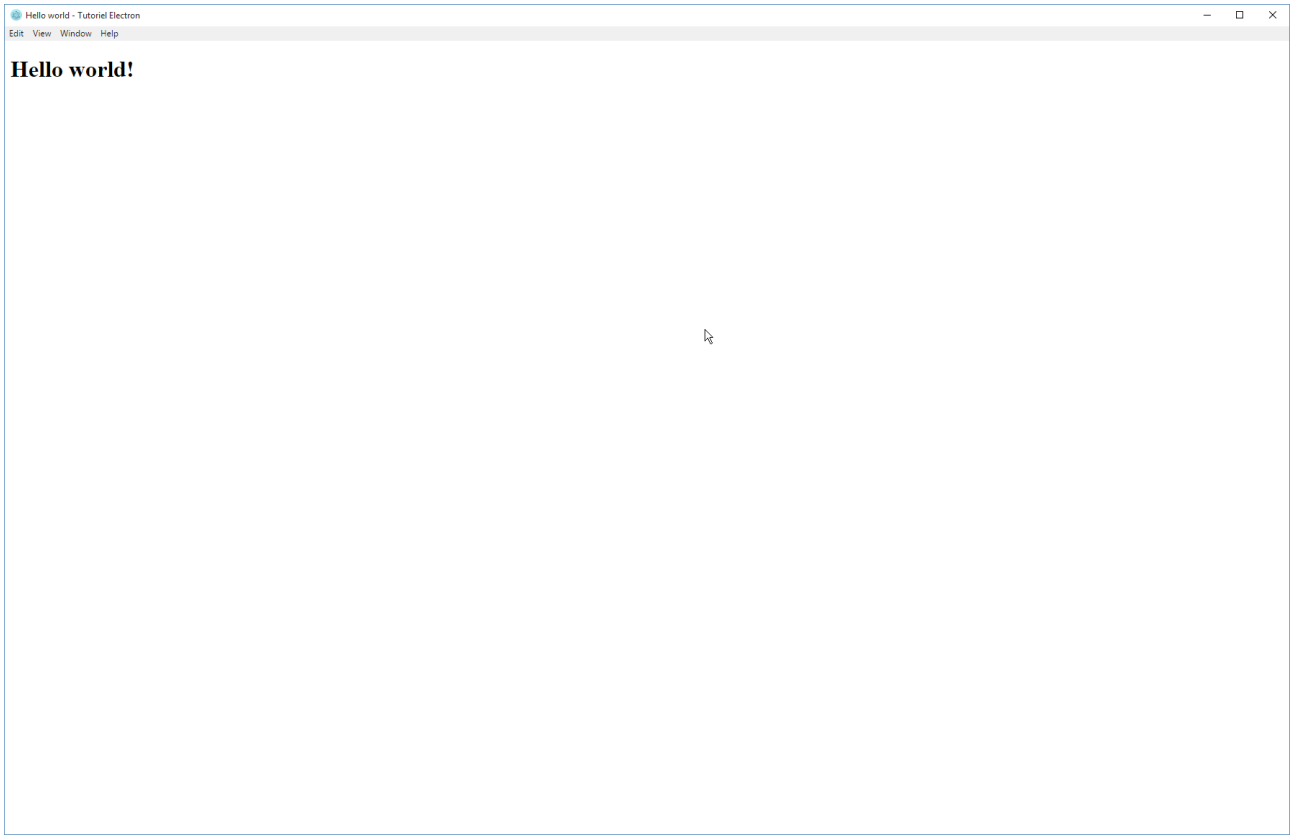

FIGURE 2. – Notre première application !

Vous pouvez utiliser les outils pour développeurs pour vous aider à déboguer vos applications. Pour les ouvrir, allez dans le menu *View*, puis cliquez sur *Toggle Developer Tools*.

3. Gérer les fenêtres

Les applications de bureau contrairement aux sites web ont une (ou plusieurs) fenêtre qui leur est dédiée. Si vous venez du monde du Web, vous ne savez donc pas comment gérer vos fenêtres avec Electron. Ça tombe bien, on va tout de suite voir comment faire.

3.1. Ouvrir des fenêtres

Dans notre première application, nous avons déjà créé une fenêtre très simple. Nous avions pour cela utilisé la classe `BrowserWindow` dont nous avons créé une instance, avec seulement deux paramètres qui définissaient la taille de la fenêtre. On peut heureusement aller beaucoup plus loin dans la personnalisation de nos fenêtres grâce à de nombreux paramètres. En voici quelques-uns parmi les plus intéressants[1].

— `width` et `height` qu'on a déjà rencontré. Ils indiquent respectivement la largeur et la hauteur de la fenêtre, en pixels.
— `x` et `y`, qui servent à définir la position d'origine de la fenêtre. Si vous voulez la centrer, vous pouvez aussi définir l'argument `center` à `true`.
— `fullscreen`, un booléen qui indique si la fenêtre est en plein écran ou non.
— `title`, simplement le titre de notre fenêtre. Si vous précisez un autre titre avec un `<title>` dans votre HTML, c'est ce dernier qui sera utilisé.
— `icon`, le chemin vers l'image qui doit servir d'icône à cette fenêtre.
— `closable`, qui est un booléen indiquant si on peut fermer la fenêtre. Si vous l'utilisez, proposez un autre moyen pour fermer votre fenêtre.
— `movable`, un autre booléen qui indique si notre fenêtre peut être déplacée ou non.

Tous ces arguments sont passés au constructeur de la fenêtre dans un objet.

```
mainWindow = new BrowserWindow({
    width: 1800,
    height: 1200,
    icon: 'assets/zds.png',
    title: 'ZdS, ça pulpe !',
    movable: false
});
```

Ici, on crée une fenêtre de 1800 pixels par 1200, avec pour icône le fichier `assets/zds.png`, un titre «ZdS, ça pulpe !» et qu'on ne peut pas déplacer.

3.1.1. Les fenêtres enfants

Les fenêtres enfants sont des fenêtres qui dépendent d'une autre fenêtre, sa parente. Elle se fermera si leur parente est fermée et elle s'affichera toujours par-dessus sa parente.

Pour en créer une, il suffit de définir le paramètre `parent` de la fenêtre qui sera l'enfant, avec pour valeur la fenêtre parent.

```
mainWindow = new BrowserWindow({title: 'Hello world !'});
const childWindow = new BrowserWindow({parent: mainWindow});
```

Vous pouvez aussi utiliser le booléen `modal`, pour qu'on ne puisse plus utiliser la fenêtre parente tant que la modale est ouverte.

3.2. Manipuler les fenêtres

Il existe de très nombreuses méthodes pour gérer une fenêtre. Vous pouvez par exemple utiliser `close` pour la fermer, `maximize` pour l'agrandir ou encore `minimize` pour la réduire (aucune de ces trois méthodes ne prennent d'arguments).

Pour jouer avec les propriétés d'une fenêtre, il faudra utiliser `getPropriete` sauf pour les booléens pour lesquels ont utilise `isPropriete`, et `setPropriete` pour définir une nouvelle valeur (avec comme argument cette valeur). Ici `Propriete` est la propriété que l'on veut.

On peut par exemple inverser le mode plein écran avec ce code.

```
1 fenetre.setFullscreen(!fenetre.isFullscreen);
```

Pour charger une page HTML, on utilise la méthode `loadURL`. On peut utiliser le protocole `file://` pour ouvrir un fichier local, mais il faut toujours que l'URL soit absolue. On va donc souvent faire quelque chose avec `__dirname`, comme ceci.

```
1 fenetre.loadURL(`file://${__dirname}mapage.html`);
```

On peut aussi définir une barre de progression, qui s'affichera avec l'icône de l'application dans la barre des tâches, avec la méthode `setProgressBar`. Cette méthode prend un nombre compris entre 0 et 1, indiquant la progression de la barre.

3.3. Les fenêtres sans bordures

Parfois, vous pouvez avoir besoin de créer une fenêtre sans bordures, par exemple pour afficher un écran de chargement si votre application est un peu lente à s'afficher, ou pour avoir une fenêtre 100% personnalisée.

Pour créer ce genre de fenêtres avec Electron, il faudra utiliser le paramètre `frame` du constructeur de `BrowserWindow`. En le mettant à `false`, notre fenêtre n'aura plus de bordures.

FIGURE 3. – Une fenêtre sans bordure

Pour obtenir ce résultat, j'ai utilisé ce code.

```
mainWindow = new BrowserWindow({
    width: 800,
    height: 500,
    icon: 'assets/zds.png',
    title: 'ZdS, ça pulpe !',
    maximized: false,
    center: true,
    frame: false
});
```

On peut même rendre notre fenêtre transparente, en utilisant le booléen `transparent`.

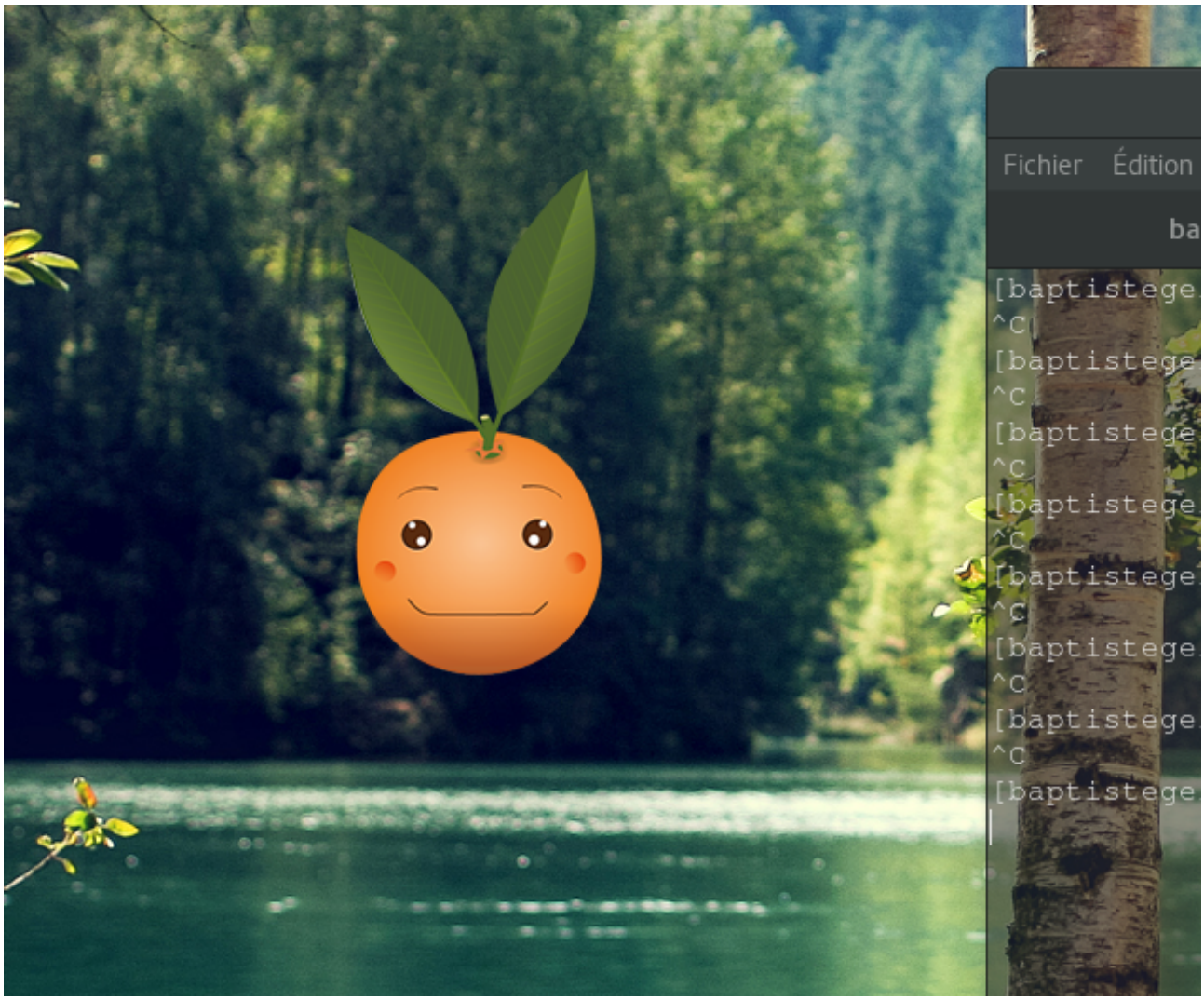

FIGURE 3. – Salut toi !

Si les fenêtres transparentes ne marchent pas et que vous êtes sous Linux, lancez votre application avec cette commande : `electron . --enable-transparent-visuals --disable-gpu`.

Pour obtenir ce résultat, j'ai simplement utilisé ce code.

```
mainWindow = new BrowserWindow({
    width: 800,
    height: 500,
    icon: 'assets/zds.png',
    title: 'ZdS, ça pulpe !',
    maximized: false,
    center: true,
    movable: false,
    frame: false,
```

```
10      transparent: true
11  });
```

J'ai aussi ajouté ce petit bout de CSS qui me permet de déplacer la fenêtre, même si elle n'a pas de bordure.

```
1  body {
2     -webkit-app-region: drag;
3  }
```

Si vous êtes sous Mac OS X, vous avez aussi la possibilité de ne pas avoir la bordure classique, tout en gardant le contour, l'ombre et les «feux tricolores» d'une fenêtre classique. Il faudra utiliser le paramètre `titleBarStyle` et le définir à `hidden`.

4. Communiquer entre les processus

Dans Electron, il existe deux types de scripts : le script principal, et les scripts de « rendu », ceux qui sont inclus dans les pages HTML. Ils peuvent tous les deux accéder aux modules Node.js classiques, mais certains modules d'Electron sont réservés au script principal, comme celle pour afficher les boîtes de dialogue (messages d'erreur, ouverture de fichier, sauvegarde …).

Pour remédier à cela, Electron nous propose deux solutions. La première, c'est l'IPC, et la seconde, l'objet `remote`.

4.1. L'IPC

L'IPC va nous permettre d'envoyer des messages entre les différents processus de notre application, c'est-à-dire le script principal et les scripts de « rendu ».

On va par exemple imaginer que vous voulez écrire quelque chose dans votre terminal depuis votre fenêtre (pour faire des logs d'erreur par exemple). Pour cela vous pouvez utiliser la méthode `console.log` depuis votre script principal. Par contre si vous appelez cette même méthode dans votre fenêtre, elle affichera le message dans la console des outils pour développeur. Il va donc falloir utiliser IPC pour dire à notre script principal de gentiment afficher un message pour nous. Il faut pour cela utiliser la méthode `ipc.send('un-signal')`. On peut donc utiliser un code similaire à celui-ci :

```
1  const ipc = require('electron').ipcRenderer;
2
3  document.getElementById('ipc').addEventListener('click', () => {
```

1. Vous pouvez retrouver la liste complète ici .

```
    ipc.send('log-error');
});
```

Il ne faut pas oublier d'ajouter un bouton (`button`) dans notre page HTML avec pour identifiant `ipc`, ainsi qu'un script pour mettre notre code JavaScript (mettez-le à la fin de votre page, pour éviter d'avoir à attendre que le DOM soit prêt).

Ensuite dans notre script, on importe l'object `ipcRenderer` du module `electron`. C'est lui qui va nous servir à utiliser IPC dans notre script de rendu. Lors du clic sur le bouton, on enverra un signal au script principal *via* IPC pour lui dire d'écrire un message dans le terminal, grâce à la méthode `ipc.send`. En effet, pour communiquer avec IPC, on a différents signaux qui peuvent être écoutés ou envoyés, et chacun à un nom. Ici on utilise donc un signal qui s'appelle `log-error`.

Maintenant que notre script de rendu envoie un signal, il faut le récupérer. Pour cela, il faut aller dans notre script principal et importer l'objet `ipcMain` du module Electron. Ensuite, on écoute l'évènement `log-error` comme n'importe quel autre, avec la méthode `on`. Et on y met le code qui affiche une erreur dans le terminal.

```
// Les requires du début …

const ipc = electron.ipcMain;

// Le reste de notre code (ouverture de fenêtre, etc) …

ipc.on('log-error', () => {

        console.log('Erreur ! Veuillez rapporter ce bug au développeur de l\'ap
});
```

Si vous lancez votre application et que vous cliquez sur votre bouton, un message devrait s'afficher dans le terminal.

Mais IPC permet aussi de faire passer des arguments entre les différents processus. Ainsi, on pourrait avoir un message d'erreur personnalisé depuis la fenêtre, sans avoir à créer deux signaux IPC différents.

Pour passer des arguments entre les processus, il suffit de rajouter un argument à la méthode `ipc.send`, qui est l'objet que vous voulez faire passer.

```
ipc.send('log-error', 'Fichier introuvable');
```

Ensuite, dans le callback du script principal, il faut rajouter deux arguments : `event` qui correspond à l'évènement et `arg` qui correspond à l'argument qu'on a fait passer.

```
ipc.on('log-error', (event, arg) => {

        console.log(`Erreur (${arg}) ! Veuillez rapporter ce bug au développeur
});
```

Ici, j'ai envoyé du texte, mais vous pouvez envoyer des nombres, des objets, des tableaux …

Vous pouvez aussi répondre à un message, grâce à la méthode `event.sender.send`. Elle s'utilise de la même façon que `ipc.send`. Il faudra ensuite écouter pour le signal dans le script de votre fenêtre. On peut ainsi envoyer la confirmation que notre message a bien été envoyé.

La communication avec IPC marche aussi dans l'autre sens : vous pouvez envoyer des signaux depuis le script principal, avec la même fonction `send` (qui se trouve dans l'objet `webContents` des instances de `BrowerWindow`) et les recevoir dans le script de votre page HTML.

On va donc écouter un signal `error-logged` dans notre script de rendu et avertir l'utilisateur grâce à un `alert`.

```
ipc.on('error-logged', () => {

        alert('Une erreur a été rencontrée. Consultez le terminal pour plus de
});
```

Il ne nous reste plus qu'à envoyer ce signal depuis le script principal.

```
// À la suite de l'affichage de l'erreur …
event.sender.send('error-logged');
```

Voilà, on a vu les possibilités d'IPC, on va maintenant étudier l'objet `remote`.

4.2. remote : accéder à toutes les APIs d'Electron dans les scripts de rendu

L'objet `remote` nous permet d'utiliser toutes les APIs normalement indisponibles dans les scripts de rendu, comme l'API *Tray* qui permet d'afficher des icônes dans la barre de notification (nous verrons comment l'utiliser). On peut obtenir cet objet très facilement, car il est dans le module `electron`. On va par exemple pouvoir afficher une boîte de dialogue, avec ce code.

```
1 const remote = require('electron').remote;
2
3 // L'objet dialog n'est normalement pas accessible depuis les
      scripts de rendu.
4 remote.dialog.showErrorBox('Erreur !',
      'L\'application a rencontré une erreur. Votre ordinateur va s\'auto-détrui
```

FIGURE 4. – L'erreur s'affiche bien

Et voilà, vous savez faire communiquer vos différents processus Electron entre eux.

5. Les boîtes de dialogue

Vous venez de voir comment afficher un message d'erreur, mais Electron propose bien plus que ça. On va donc découvrir comment ouvrir des boîtes de dialogue de sélection de fichier, d'information, de sauvegarde.

5.1. Les messages d'information, d'erreur, de question, et compagnie

Pour afficher des messages d'erreur, nous l'avons vu juste avant, on peut utiliser la méthode `showErrorBox` de l'objet `dialog` (qui fait partie du module `electron`, et qu'on a utilisé *via* l'objet `remote`). Cette fonction prend juste deux arguments :un titre, et un message.

Mais on peut aussi afficher des messages qui ne sont pas des erreurs. Pour cela on utilise la méthode `dialog.showMessageBox`. Cette méthode prend comme argument un objet, qui représente les options à utiliser pour notre boîte de dialogue. Parmi elles, on trouve …

— `type`, qui peut être soit `'none'`, `'info'`, `'error'`, `'question'`, `'warning'` ou `'info'`. Vous l'avez deviné c'est le type de la boîte de dialogue à afficher, et donc l'icône qui va avec ;

— `title`, qui est le titre de la boîte de dialogue ;
— `message`, qui est le message à afficher ;
— `buttons`, un tableau contenant les différents boutons sur lesquels l'utilisateur pourra cliquer.

Mais il y en a beaucoup plus, je vous laisse regarder dans la documentation . Par exemple, on peut utiliser ce code …

```
remote.dialog.showMessageBox({
    type: 'warning',
    title: 'Attention !',
    message:
        'Faites très attention. C\'est très dangereux par ici.',
    buttons: ['D\'accord',
        'Euh ... je vais faire demi-tour alors.']
});
```

… et voir apparaître ceci.

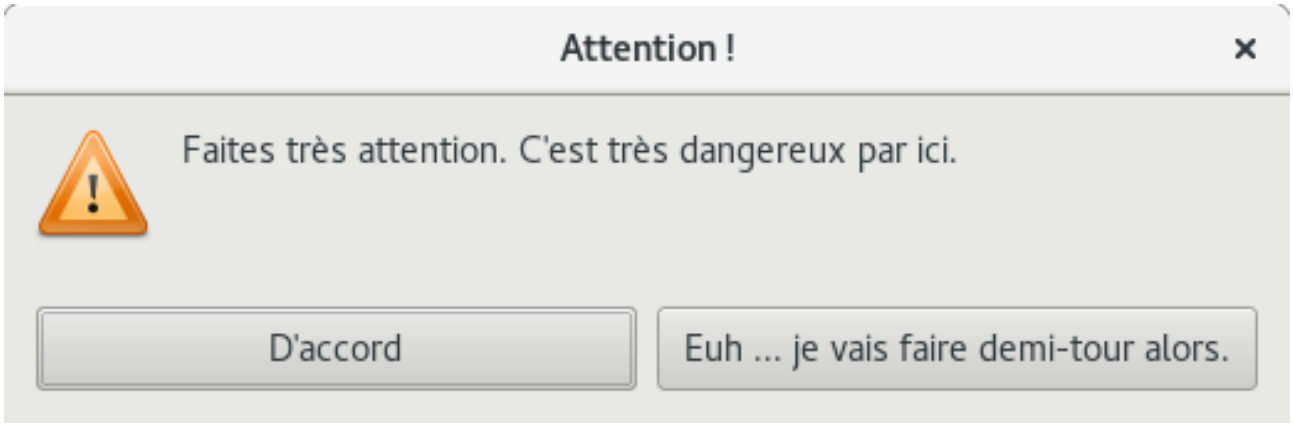

FIGURE 5. – Attention !

5.2. Enregistrer et ouvrir des fichiers

La plupart des applications de bureau servent à modifier des fichiers, que ce soit du texte, des images, des vidéos ou autre. Savoir ouvrir et enregistrer des fichiers est donc essentiel.

Dans Electron, vous pouvez demander à l'utilisateur où il veut enregistrer son travail grâce à des boîtes de dialogue (comme dans beaucoup d'autres applications en fait).

Pour savoir où enregistrer un document, il faut utiliser la méthode `dialog.showSaveDialog`. Cette méthode prend un objet qui représente ces options. On peut y mettre ces propriétés :

— `title`, le titre de la boîte de dialogue ;
— `defaultPath`, le chemin à ouvrir par défaut ;
— `buttonLabel`, les textes à afficher sur les boutons. C'est un tableau de deux éléments ;

— `filters`, les filtres.

Les filtres précisent les types de fichiers que l'utilisateur peut choisir. C'est une liste de plusieurs objets qui ont cette forme.

```
{name: 'Musique', extensions: ['mp3', 'ogg', 'wav']}
```

Pour récupérer ce que l'utilisateur a choisi comme emplacement pour sa sauvegarde et donc pouvoir y écrire ce qu'on veut, il faut utiliser un callback. Seulement si on regarde bien la documentation, il faut alors aussi préciser une instance de la classe `BrowserWindow`. Pour obtenir la fenêtre actuelle, lorsque vous utilisez cette fonction *via* `remote`, vous pouvez faire `remote.getCurrentWindow()`, sinon vous pouvez simplement utiliser la variable `mainWindow`. Le callback quant à lui, prend un argument qui est le chemin vers le fichier sélectionné. Ce code affichera donc une fenêtre pour sauvegarder des fichiers et affichera le chemin sélectionné dans la console.

```
remote.dialog.showSaveDialog(remote.getCurrentWindow(), {
    title: 'Enregistrez votre travail',
    filters: [
            {name: 'Fichiers de texte brut', extensions: ['txt']},
            {name: 'Tous les fichiers', extensions: ['*']}
        ]
    },
    (file) => {
        console.log(`Vous avez enregistré sous ${file}.`);
    });
```

Pour ouvrir des fichiers, il faut utiliser la fonction `dialog.showOpenDialog`. Cette fonction ressemble beaucoup à celle pour enregistrer, mais peut prendre une option en plus. C'est le tableau `properties`, qui peut contenir une des valeurs suivantes :

— `'openFile'`, à mettre si vous voulez que l'utilisateur sélectionne des fichiers ;
— `'openDirectory'`, si vous voulez qu'il sélectionne des dossiers ;
— `'multipleSelections'`, s'il a le droit de sélectionner plusieurs fichiers ou dossiers ;
— `'createDirectory'`, s'il a le droit de créer un nouveau dossier ;

6. Les menus

Les menus contextuels existent dans la spécification HTML5, mais ne sont supportés que par Firefox aujourd'hui. Electron a donc sa propre API pour les menus, qui peut en plus des menus contextuels, afficher des menus pour la fenêtre en général.

6.1. Afficher un menu contextuel

Il faut savoir qu'il existe deux méthodes pour créer des menus (contextuels ou pas) dans Electron. La première est principalement basée sur la fonction `append` de la classe `Menu`, mais ça devient souvent assez difficile à comprendre quand on commence à avoir des menus un peu gros. La seconde, que je vous montrerai dans ce tutoriel, se base sur des templates, c'est-à-dire un objet, ou plutôt un tableau d'objets, qui va décrire notre menu.

Chaque item de notre menu sera un objet pouvant avoir ces propriétés :

- `label`, qui représente le texte affiché ;
- `icon`, qui est un chemin vers l'icône du menu ;
- `accelerator`, un raccourci clavier pour cette action. Nous verrons plus tard comment l'utiliser ;
- `type`, qui peut être soit `'normal'`, `'separator'`, `'submenu'`, `'checkbox'` ou `'radio'`. Je pense que vous devinez à quoi correspondent chacune des valeurs ;
- `submenu`, qui est un tableau contenant les éléments d'un sous-menu ;
- `click`, qui est une fonction à appeler lorsqu'on clique sur cet élément.

Il y a d'autres options disponibles, elles sont moins importantes. Je vous laisse les découvrir par vous-même si vous voulez.

Pour créer un menu depuis un template, il faut utiliser la fonction `Menu.buildFromTemplate` à laquelle on passe notre template. Ensuite, on utilise la méthode `unMenu.popup` pour afficher notre menu.

```
1  const Menu = require('electron').remote.Menu;
2
3  // On considère qu'on a une <div id="menu"> dans notre HTML
4
5  document.getElementById('menu').addEventListener('contextmenu',
       (event) => {
6      event.preventDefault();
7      const template = [
8          {
9              label: 'Ajouter aux favoris',
10             click: () => {
11                 // TODO : ajouter aux favoris
12                 alert('Article bien ajouté aux favoris');
13             }
14         },
15         {
16             label: 'Partager',
17             submenu: [
18                 {
19                     label: 'Diaspora*',
20                     icon: 'assets/diaspora.png',
21                     click: () => {
22                         shareOnDiaspora();
23                     }
```

```
                },
                {
                    label: 'GNU Social',
                    icon: 'assets/gnusocial.png',
                    click: () => {
                        shareOnGnuSocial();
                    }
                }
            ]
        }
    ];
    const menu = remote.Menu.buildFromTemplate(template);
    menu.popup();
});
```

Bien-sûr on suppose qu'on a une fonction `shareOnDiaspora` et une autre `shareOnGnuSocial` dans notre code, et qu'on a bien nos icônes dans le dossier *assets*.

Si vous utilisez des icônes, pensez à en faire de petite taille (16x16 pixels), car elles ne seront pas redimensionnées automatiquement et vous pourriez vous retrouver avec des icônes énormes.

6.2. Définir un menu pour son application

On a vu comment définir un menu contextuel pour un élément. Mais certaines actions doivent pouvoir être accessibles à tout moment, comme l'enregistrement, les options, l'aide ou les classiques *Annuler* et *Refaire*.

Pour cela, la meilleure solution est de passer par un menu visible à tout moment en haut de la fenêtre (sous Windows et la plupart des Linux) ou dans la barre de notifications (sur Mac OS X et certains Linux).

En définissant votre propre menu, vous n'aurez plus accès à celui par défaut et donc aux outils de développement. Pour éviter cela, vous pouvez utiliser la méthode `webContents.openDevTools` de la fenêtre que vous manipulez.

Electron nous permet de faire ça grâce à la méthode `Menu.setApplicationMenu` qui demande bien évidemment un menu comme argument.

```
const template = [
    {
        label: 'Fichier',
        submenu: [
```

```
            {
                label: 'Enregistrer'
            }, {
                label: 'Ouvrir'
            }, {
                label: 'Quitter'
            }
        ]
    },
    {
        label: 'Édition',
        submenu: [
            {
                label: 'Annuler'
            }, {
                label: 'Refaire'
            }, {
                label: 'Copier'
            }, {
                label: 'Coller'
            }
        ]
    }
];
const menu = remote.Menu.buildFromTemplate(template);
Menu.setApplicationMenu(menu);
```

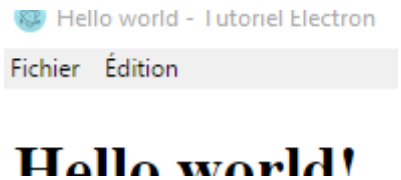

FIGURE 6. – Notre menu s'affiche bien

Cependant, cette API est assez lente quand on l'utilise avec `remote`, et comme ce menu sera disponible dans toute votre application, il vaut mieux l'utiliser directement dans le script principal.

i Sous Linux et Windows, vous pouvez aussi définir un menu qui sera affiché dans une fenêtre seulement, grâce à la méthode `fenetre.setMenu(monMenu)`.

6.3. Les raccourcis-clavier

Pour gagner du temps, rien de tel que des raccourcis clavier. Electron nous propose une syntaxe très simple pour en créer. Il suffit en effet de créer un `String` contenant les différentes touches

de notre raccourci, séparées par des `+`. Par exemple, pour le raccourci Ctrl+C, vous n'aurez qu'à faire `Ctrl+C`.

> Les Macs n'ont pas de touche Ctrl, mais un équivalent qui est Cmd. Electron nous permet de détecter l'un ou l'autre grâce à `CmdOrCtrl`.

Comme on l'a vu plus haut, associer un raccourci-clavier à un élément d'un menu peut se faire simplement en définissant la propriété `accelerator` dans le modèle de notre menu.

```
{
    label: 'Copier',
    accelerator: 'CmdOrCtrl+C'
}
```

7. La barre de notifications

Pouvoir notifier ces utilisateurs, ou bien simplement fournir un raccourci vers notre application même quand elle tourne en tâche de fond est toujours utile. Grâce à la classe `Tray`, Electron nous permet de réaliser ceci.

FIGURE 7. – Des icônes dans la barre de notification. Bientôt, la vôtre sera là aussi.

Cette classe n'est utilisable que dans le processus principal, à moins de passer par `remote`. Pour afficher votre icône dans la barre de notifications, il faut créer une nouvelle instance de la classe `Tray`, qui se trouve encore une fois dans le module `electron`. Son constructeur prend pour argument le chemin vers l'icône qu'on veut afficher.

```
// On a les mêmes require que tout à l'heure …

const Tray = electron.Tray;

function createWindow () {

    const tray = new Tray('assets/zds.png');
    // Petit bonus : on affiche une bulle au survol.
    tray.setToolTip('Zeste de Savoir');
    // Notre fichier continue avec l'initialisation de la fenêtre,
        etc.
```

Comme beaucoup d'autres APIs Electron, on ne peut pas utiliser la classe `Tray` avant l'évènement `ready` d'`app`. Avant cet évènement, Electron n'est pas complètement initialisé.

FIGURE 7. – Notre icône est bien là.

7.1. Interagir avec l'utilisateur

Bon, on a une jolie icône, mais si on la rendait utile ? Par exemple, si on lui ajoutait un petit menu contextuel ?

On va pour cela utiliser la fonction `setContextMenu` de notre constante `tray` qui, bien entendu, prends un menu comme argument (ça tombe bien on a appris à en créer juste avant).

```
1  const menu = Menu.buildFromTemplate([
2      {
3          label: 'Ouvrir les forums',
4          click: () => {
5              // cette méthode permet d'ouvrir une URL dans le
                   navigateur par défaut
6
                   electron.shell.openExternal('https://zestedesavoir.com/forums')
7          }
8      }
9  ]);
10 tray.setContextMenu(menu);
```

7.2. Envoyer une notification

En plus d'afficher une petite icône dans la barre de notification, on peut envoyer des notifications à l'utilisateur. On utilise pour cela une fonctionnalité standard (bien qu'encore à l'état de brouillon) : la classe `Notification`.

On ne peut pas afficher de notifications avec cette méthode sous Windows 7 et inférieur. On peut cependant utiliser le `Tray`, *via* sa fonction `displayBalloon` .

En créant une nouvelle instance de la classe `Notification`, une nouvelle notification sera automatiquement affichée. Le constructeur demande deux arguments : un titre, et des options optionnelles sous la forme d'un objet. Parmi elles on a notamment `body`, qui est le corps de la notification où on va afficher des détails et `icon`, un chemin vers l'icône à utiliser.

```
new Notification('Une réponse a été postée !', {
    body: 'Clem dans le sujet «Electron, c\'est bien.»,'
    icon: 'assets/zds.png'
});
```

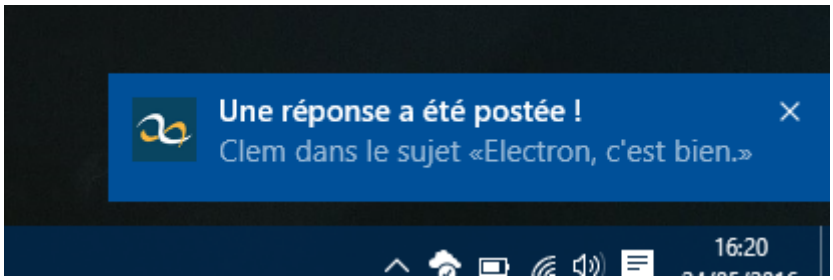

FIGURE 7. – Oh ! Une notification !

Et voilà, vous savez comment spammer vos utilisateurs de notifications.

8. Publier son application

Vous venez de terminer la première version de votre fantastique application, et vous aimeriez la distribuer ? Ça tombe bien, c'est très simple avec Electron.

En effet, il existe un petit programme en ligne de commande appelé `electron-packager` qui vous permet de créer des fichiers exécutables pour Mac OS X, Windows et Linux, quel que soit votre système d'exploitation !

C'est un package npm, on va donc commencer par l'installer *via* notre terminal.

```
npm install electron-packager -g --save-dev
```

i Là aussi, pour éviter des erreurs, pensez à passer en mode *Administrateur* ou en *root*.

Ensuite, vous n'avez plus qu'à lancer ceci pour « compiler » votre application (en admettant que vous êtes bien dans le dossier du code de votre application).

```
electron-packager . --all
```

Ici, l'option `--all` permet d'empaqueter pour toutes les plateformes et architectures. Une fois que la commande est lancée, vous n'avez plus qu'à attendre, les différentes versions d'Electron pour chacune des plateformes vont être téléchargées et votre application empaquetée.

Si vous obtenez une erreur comme `Unable to infer name or version. Please specify a name and version.`, exécutez plutôt la commande suivante : `electron-packager . --all --name "Un super nom" --version "1.0.0"`.

Si vous êtes sous Linux ou Mac OS X, il faut savoir que vous devrez d'abord installer Wine pour pouvoir obtenir une version Windows de votre application.

Une fois que vos fichiers exécutables ont étés créés, vous pouvez en faire profiter tout le monde en les postant sur votre site web, ou tout autre système de partage de fichier (cloud et autres). Vous pouvez aussi utiliser des outils comme electron-deb qui permet de créer un fichier .deb avec votre application pour ensuite la rendre facilement installable aux utilisateurs d'APT (Debian, Ubuntu...). Il existe aussi electron-installer qui peut générer des installateurs Windows pour votre application.

Voilà, ce tutoriel est terminé. J'espère que vous vous amusez bien avec Electron.

Si jamais vous avez un problème, les forums sont là pour ça ! (Mais pensez à faire une petite recherche avant quand même). La documentation officielle peut aussi vous aider, ce tutoriel ne présentant que les APIs les plus importantes.

Si jamais vous avez des questions, n'hésitez pas à créer un sujet sur le forum, avec le tag « Electron », vous trouverez sûrement des réponses à vos questions (mais pensez quand même à faire une petite recherche avant). Et si vous créez une application avec Electron, n'hésitez pas non plus à la présenter dans le forum dédié.

Je voudrais quand même remercier La Source, WinXaito, Demandred et victor qui ont participé à la bêta du tutoriel, ainsi que tcit qui a validé ce tutoriel !

Liste des abréviations

IPC Inter Process Communication. 1, 12–14