Les évènements personnalisés

Temps de lecture : 6 minutes

Fonctionnement des `eventEmitter`

Avant de voir le module `core` proposé par `Node.js`, nous allons créer notre propre module.

En tant que développeur JS nous avons l'habitude des évènements : une touche pressée, un clic sur un élément HTML : ce sont des évènements.

Nous sommes très habitués aux architectures orientées évènements.

Nous enregistrons des écouteurs, *listeners* et leur passons en `callback` une fonction appelée gestionnaire d'évènement, ou *event handler*.

Avec `Node.js` c'est exactement la même chose, sauf que nous pouvons créer nos propres évènements qui peuvent être déconnectés de toute interaction utilisateur.

Création de notre `eventEmitter`

Le module proposé par `Node.js` est super robuste et permet de gérer toutes les erreurs possibles.

Avant de l'utiliser nous allons tout simplement créer notre `eventEmitter`, son fonctionnement sera très similaire à celui de `Node.js` mais sera moins robuste.

Etape 1 : notre `Emitter`

Nous allons tout simplement créer un fichier `Emitter.js`.

Notez la majuscule qui est une convention de nommage lorsqu'il s'agit d'une classe.

```
module.exports = class Emitter {
  constructor() {
    this.events = {}
  }
}
```

Nous exportons une classe `JavaScript` et nous définissons un `constructor` qui va permettre d'initialiser une propriété `events` qui sera un objet vide pour chaque nouvelle instance de notre `Emitter`.

Une instance est créée en faisant :

```
const emitter = new Emitter();
```

Nous plaçons toujours dans notre classe une méthode :

```
on(type, listener) {
  this.events[type] = this.events[type] || [];
  this.events[type].push(listener);
}
```

Cette méthode sera sur le prototype de chaque instance de notre classe.

Que fait-elle ?

Elle a deux paramètres, le premier est le `type` d'évènement qui sera seulement une chaîne de caractères du nom de notre évènement.

Par exemple : `utilisateur-connecte`, `fileopened`, bref ce que vous voulez.

Le second sera une fonction, notre fameux `listener` ou `event handler`.

Notre objet `events` de notre instance contiendra donc un tableau pour chaque `type` d'évènement.

Ce tableau contiendra tous les `listeners` créés avec notre méthode `on(type, listener)` pour un `type` d'évènement particulier, ce sera donc un **tableau de fonctions**.

Par exemple :

```
{
  user-connected: [
    sendMessagesToAllConnectedUsers() {...},
    incrementConnectedUsers() {...}
  ]
}
```

Que fait cette ligne ?

```
this.events[type] = this.events[type] || [];
```

Nous nous assurons juste qu'il existe un tableau de `listeners` pour un évènement donné sur l'objet `events`, et s'il n'y en a pas nous en créons un.

Nous créons une deuxième méthode sur notre classe :

```
emit(type) {
  if (this.events[type]) {
    this.events[type].forEach(listener => listener());
  }
}
```

Cette fonction va nous permettre d'exécuter tous les `listeners` définis lorsque le `type` d'évènement auquel elles sont attachées est envoyé avec la méthode `emit`.

Etape 2 : nos `events`

En allant un tout petit peu plus loin que la vidéo, nous allons en profiter pour vous montrer une bonne pratique très classique.

Pour les évènements, il est d'usage de créer un objet liant une chaîne de caractères à un nom de propriété écrit en capitales.

Premièrement, pour ne pas faire d'erreur en écrivant l'évènement, cela saute plus aux yeux.

Deuxièmement, parce que votre éditeur de code est génial et vous permettra d'utiliser l'autocomplétion : vous n'aurez qu'à taper le début du nom puis entrée.

Nous créons donc un fichier `events.js` et nous exportons juste un objet avec ces liaisons :

```
module.exports = {
  FILE_OPENED : 'fileopened',
  WRITTING_FILE : 'filewritting',
  FILE_CLOSED : 'fileclosed'
}
```

Etape 3 : exemple d'utilisation

Nous allons importer nos deux modules et les utiliser :

```
const Emitter = require('./Emitter');
const events = require('./events');

const emitter = new Emitter();

emitter.on(events.FILE_OPENED, () => console.log('Fichier ouvert'));
emitter.on(events.FILE_OPENED, () => console.log('Fichier ouvert 2'));

emitter.emit(events.FILE_OPENED);
```

Nous créons deux `listeners` qui vont seulement `console.log` un message lorsqu'un évènement de type `'fileopened'` sera émis.

Nous émettons ensuite un évènement de type `events.FILE_OPENED` ce qui va entraîner l'exécution de nos deux `listeners`.

Et… c'est tout :)

Vous savez maintenant comment fonctionne ce pattern de programmation extrêmement courant aussi bien en front qu'en back.
https://repl.it/@dymafr/node-c3-l3

Dans la prochaine leçon nous allons étudier le module `core` de `Node.js`.