Module Event

Les API de base de Node.js sont construites autour d'une architecture événementielle asynchrone.

L'asynchronisme est mis en place avec des émetteurs et des auditeurs d'événements.

Ainsi de très nombreux modules héritent de **Class**[1]**: `EventEmitter`**

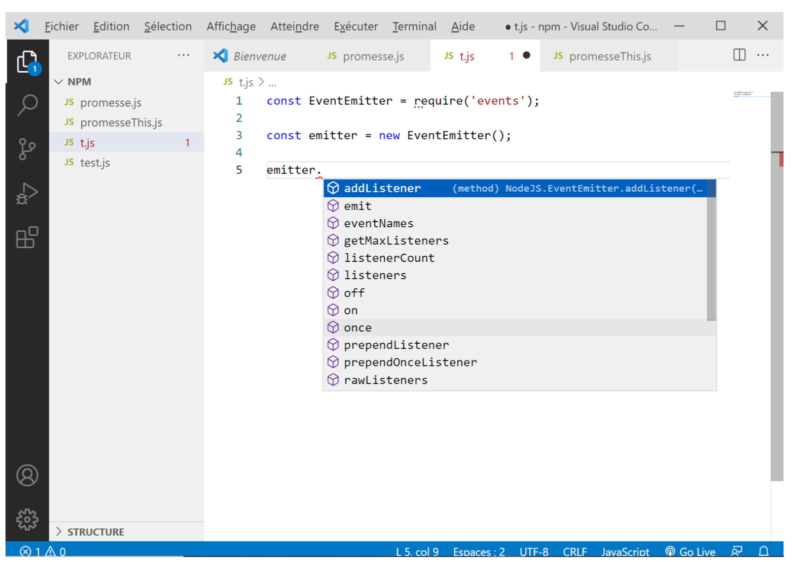
1. const EventEmitter[2] = require('events');

Une classe est un conteneur pour un grand nombre de propriétés et méthodes. Pour utiliser ces méthodes, il convient de créer une instance de la classe. Ainsi dans le code suivant, EventEmitter est la classe et emitter une instance.

1. const EventEmitter = require('events');
2. const emitter = new EventEmitter();

Lig. 2 : emitter est un objet de la classe EventEmitter.

Le nombre de méthodes associées à l'objet **emitter** est important. Il suffit de compléter le code **emitter.**⭾ (tabulation) pour le constater dans la figure suivante[3].



[1] cours : class in ES6
[2] Nous notons que par convention le nom d'une classe commence par une majuscule.

[3] On retrouve dans cette liste le célèbre addEventListener.

Nous nous limitons cependant à deux méthodes ***emit*** et ***on***. Nous allons revenir sur ces deux méthodes en détail par la suite.

Emit

La méthode **emit** appelle l'écouteur enregistré pour l'événement nommé par le paramètre.

Ecrivez le code suivant dans le fichier debug.js

debug.js

```
1. const EventEmitter = require('events');
2. const emitter = new EventEmitter();
3.
4. emitter.emit('messageLogged')
```

Lig. 4 : On signale qu'un message est enregistré. L'idée est donc de faire réagir les écouteurs du signal **'messageLogged'.**

Exécutez le code de debug.js

```
C:\Users\DD\Desktop\adetruire>node debug.js
```

L'exécution de ce code n'est pas très concluante !

En fait, l'exécution de ce programme ne donne rien car pour l'instant nous n'avons pas d'écouteurs.

On émet un signal, mais on n'écoute pas ce signal !

On

Avec la méthode *on*, nous enregistrons un gestionnaire d'événement. On indique "quoi faire" lorsque l'on reçoit un événement.

La méthode *on* peut-être considérée comme un alias à addEventListener. Chaque événement a un gestionnaire[4] d'événement, qui est la fonction qui sera exécutée lorsque l'événement se déclenche. Nous enregistrons simplement l'événement avec la méthode *on*.

Créez un fichier event.js. Nous allons ajouter un écouteur à notre signal 'messageLogged'.

event.js

```
1.  const EventEmitter = require('events');
2.  const emitter = new EventEmitter();
3.
4.  emitter.on('messageLogged', function(){
5.      console.log(`listener called`)
6.  })
7.
8.  emitter.emit('messageLogged')
```

Lig.4 : On indique qu'il faut afficher 'listener called' si on reçoit l'événement 'messageLogger'.

Lig. 8 : On émet un signal.

En exécutant le programme event.js un message est envoyé et reçu. Cela déclenche une action d'affichage !

```
C:\Users\DD\Desktop\test>node event.js
```

L'exécution affichera "listener called"

[4] ou plusieurs

Les arguments

Il est facile de passer des arguments entre l'émetteur et les gestionnaires.

L'argument peut rassembler un grand nombre de propriétés sous la forme d'un objet.

event.js

```
1.  const EventEmitter = require('events');
2.  const emitter = new EventEmitter();
3.
4.  emitter.on('messageLogged', ( arg ) =>{
5.      console.log(`name: ${arg.name}`)
6.  })
7.
8.  emitter.emit('messageLogged', {name:"superDupont"})
```

Lig. 8 : On émet un signal avec un argument.

Lig. 4 : On récupère les arguments passés en lig.8 dans une fonction de callback.

This

Il est important de noter que dans le cas d'un cb en écriture conventionnelle (avec *function* et non une fonction fléchée), le this a l'intérieur du cb est lié à l'émetteur.

event.js

```
1.  const emitter = new EventEmitter();
2.
3.  emitter.on('messageLogged', function(arg){
4.      console.log(this == emitter)
5.      console.log(`name: ${arg.name}`)
6.  })
```

7.
8. emitter.emit('messageLogged', {name:"superDupont"})

Si nous exécutons le programme, nous pouvons vérifier que **this** est l'instance emitter et que les arguments sont correctement transmis.

`C:\Users\DD\Desktop\test>node event.js`

true

name: superDupont

Réécrivons le cb avec une fonction fléchée.

event.js

```
1. const emitter = new EventEmitter();
2.
3. emitter.on('messageLogged', (arg) => {
4.     console.log(this, this == emitter)
5.     console.log(`name: ${arg.name}`)
6. })
7.
8. emitter.emit('messageLogged', {name:"superDupont"})
```

Lig. 3 : le cb est une fonction fléchée.

L'utilisation d'une fonction fléchée comme fonction de rappel affichera :

`C:\Users\DD\Desktop\npm>node event.js`

{} false

name: superDupont

Ainsi, dans ce cas de la déclaration du cb avec une fonction fléchée, le this n'est plus lié à l'émetteur.

Heritage

En réalité, il est très rare d'utiliser EventEmitter directement comme nous venons de le faire.

En introduction, nous écrivons que les API de base de Node.js sont construites autour d'une architecture événementielle asynchrone.

Pour créer une classe capable de lever des événements, nous devons étendre EventEmitter. Pour ce faire, chaque module de node hérite de la classe EventEmitter.

Nous allons comprendre le besoin. Nous verrons ensuite la nécessité de l'héritage.

Module

Commençons par créer un simple module qui permet l'affichage et émet un message de confirmation.

Ecrivons dans un fichier logger.js le code suivant :

logger.js

```
1.  const EventEmitter = require('events');
2.  const emitter = new EventEmitter();
3.
4.  function log(message) {
5.      console.log(message);
6.
7.      //raise an event
8.      emitter.emit('messageLogged', {name : "superDupont"});
9.  }
10.
11. module.exports = log
```

node: module event p.7

Créons une application app.js qui utilise le module logger. Pour cela créez un fichier app.js avec le code suivant :

app.js

```
1.  const EventEmitter = require('events');
2.  const log = require('./logger');
3.
4.  const emitter = new EventEmitter();
5.
6.  emitter.on('messageLogged', (arg) => {
7.      console.log(`name: ${arg.name}`)
8.  })
9.
10. log('message');
```

Lig. 6 : On définit un écouteur sur l'événement "messageLogged' défini dans le module.

Nous allons exécuter le fichier app.js

```
C:\Users\DD\Desktop\npm>node app.js
```

message

Lig. 10 : log('message') va par l'intermédiaire du module logger émettre un message emitter.emit (lig. 8) de logger.js

Lig.6 : **emitter.on** ne semble pas reçevoir pas ce message 💤 !

Explication :

La raison de cette non réception est simple. Les deux instances émetteur emitter et emitter sont différentes. Les deux instances n'ont rien à voir entre elles.

Nous allons contourner ce problème en créant une classe étendant la classe events.

Extend

Nous allons voir comment hériter de la classe EventEmitter pour régler le problème précédent.

Nous modifions notre code logger.js pour étendre la classe EventEmitter.

logger.js

```
1.  const EventEmitter = require('events');
2.
3.  class Logger extends EventEmitter {
4.
5.      log(message){
6.          console.log(message);
7.          //raise an event
8.          this.emit('messageLogged', { name: "superDupont" });
9.      };
10. }
11.
12. module.exports = Logger
```

Lig. 3 : On utilise le mot clé extends de JS qui permet l'héritage d'une classe.

Lig. 5 : La méthode de prototype est définie.

Lig. 8 : this fait référence à l'instance de la classe qui appelle la méthode.

Maintenant transformons le fichier app.js en créant une instance de la classe Logger.

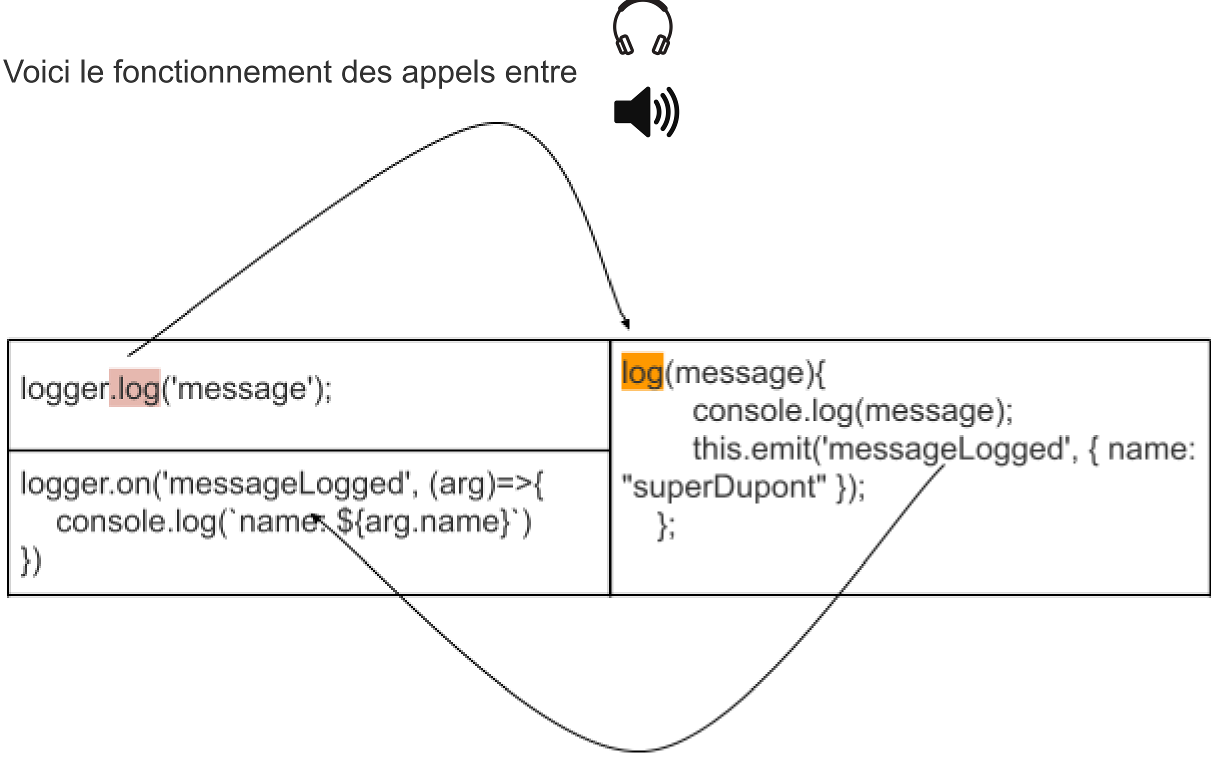
node: module event p.9

app.js

```
1. const Logger = require('./logger');
2. const logger = new Logger();
3.
4. logger.on('messageLogged', (arg)=>{
5.     console.log(`name: ${arg.name}`)
6. })
7.
8. logger.log('message');
```

Lig. 2 : Création d'une instance _logger de la classe _Logger. Notez le style d'écriture adoptée.

Voici le fonctionnement des appels entre



Voici le passage des paramètres.

<table>
<tr><td>logger.log('message');</td><td rowspan="2">log(message){
        console.log(message);
        this.emit('messageLogged', { name: "superDupont" });
    };</td></tr>
<tr><td>logger.on('messageLogged', (arg)=>{
    console.log(`name: ${arg.name}`)
})</td></tr>
</table>

node: module event p.10

Vérifions le comportement de l'exécution :

```
C:\Users\DD\Desktop\test>node event.js
```

message

name: superDupont

Cette technique est étendue à l'ensemble des modules de node. Ainsi les modules de base peuvent émettre et recevoir des signaux.

Par exemple, on peut regarder comment le module process étend la classe EventEmitter. https://nodejs.org/api/process.html