Programmation web
JavaScript - Applications 2 - Introduction à Node

Vincent Padovani, PPS, IRIF

Node est un environnement pour l'exécution de programmes JavaScript (ce qu'on appelle un "runtime"). Il se prête idéalement à l'écriture de serveurs web et fournit un grand nombre d'éléments prédéfinis permettant la construction et les envois asynchrones de réponses à des requêtes HTTP, l'accès à une base de données, etc., le tout avec une bonne économie de syntaxe. Son vrai défaut est, comme souvent avec ce genre d'outils, une documentation trop dense et assez mal organisée.

La programmation asynchrone est évidemment omniprésente dans Node, et les invocations de méthodes sont souvent associées à des fonctions de rappel (callback), par exemple avec un dernier argument de la forme `(err, data) => //*... *//` où `err` sera indéfini lors de l'invocation de la fonction en cas de réussite, défini sinon.

Un exemple d'usage immédiat de Node est la création d'un serveur local en seulement quelques lignes de code – nous reviendrons bien sûr sur les différents éléments de son contenu. La page sera visible sur `http://127.0.0.1:8080` ou `http://localhost:8080`, mais il faudra éventuellement configurer votre navigateur pour qu'il accepte le protocole `http`, et pas seulement `https` – noter que les messages sont affichés dans le terminal, et non dans la console du navigateur :

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 8080;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain; charset=UTF-8');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

1 Installation et usage

Sur une distribution telle qu'Ubuntu, le package `nodejs` est en principe déjà installé sur votre machine (vérifiez avec `which node`, et installez ce package si la commande est introuvable)[1]. L'exécution d'un fichier `fichier.js` utilisant les modules de `Node.js` se fait à l'aide de la commande : `node fichier.js`.

1. Des installers pour Windows et MacOS sont disponibles sur http://nodejs.org/.

2 Le système de modules de Node

En programmation Node, il est normal – et même souhaitable – de répartir les différents du code dans plusieurs fichiers. Node fournit un mécanisme permettant à chaque fichier de disposer de son propre espace de noms, sans risque que l'une des définitions d'un fichier n'ecrase une definition de même nom d'un autre fichier. Chacun de ces fichiers est appelé un *module*.

2.1 Eléments exportés

Par défaut, les éléments définis dans un module sont inaccessibles dans tout autre module, à moins qu'ils ne soient explicitement *exportés*. Chaque fichier, s'il est exécuté et au moment où il est exécuté, l'est dans un contexte où est défini un objet global propre à ce fichier et initialement vide : l'objet `exports`. L'exportation d'éléments du module se fait en ajoutant ces éléments comme de nouvelles propriétés de `exports` : les valeurs de ces propriétés deviendront accessibles à l'extérieur du module.

```
// fichier unModule.js
// ...
exports.n = 42;
exports.f = function() { /* ... */ }
exports.g = function() { /* ... */ }
// etc.
```

On peut aussi utiliser l'écriture usuelle suivante. Un autre objet global, `module`, possède quant à lui une propriété `exports`, par défaut égale à l'objet `exports`. Si l'on remplace `module.exports` par un autre objet, les éléments exportés seront toutes ses propriétés :

```
let n = 42;
let f = function() { /* ... */ }
let g = function() { /* ... */ }
module.exports = { n, f, g } // équivalent à : { n : n, f : f, g : g }
```

2.2 Eléments importés

L'import dans un module d'éléments exportés par un autre se fait à l'aide de la fonction prédéfinie `require`. Cette fonction prend en argument le nom de ce second module – l'ajout de son extension est optionnel. Elle exécute dans ce cas l'intégralité du contenu de ce module, puis renvoie un objet servant de point d'accés à ses éléments exportés :

```
// fichier autreModule.js
// ...
// exécution de unModule.js, retour d'un objet ayant
// toutes les propriétés exportées dans ce fichier :
const mod = require("./unModule");
mod.n; // => 42
```

Noter qu'il est possible d'importer plusieurs fois le même module – mais ici, pas avec la référence `mod`, dont la valeur n'est plus modifiable. Noter également que Node propose un grand nombre de modules prédéfinis : c'est le cas par exemple du module `http.js` utilisé dans le mini-serveur de l'introduction. Dans le dernier exemple, L'ajout de `"./"` avant `"unModule"` n'est pas indispensable, mais souligne simplement le fait qu'il ne s'agit pas de l'un des modules prédéfinis.

3 Les événements en Node

3.1 Émetteurs et écouteurs d'événements

Un serveur fonctionnant essentiellement de manière asynchrone, la plupart des objets impliqués dans la construction ou le fonctionnement d'un serveur Node sont des *émetteurs d'événements*, en un sens similaire à celui des événements du DOM : il existe différents types d'événements associés aux changements d'états du serveur, et l'on peut ajouter aux objets émetteurs des écouteurs pour chaque type d'événements. La syntaxe de cet ajout est toujours de la forme suivante :

```
emitter.on(eventType, listener);
```

Tous les émetteurs d'événements dans Node sont des instances de classes héritières de la classe `events.EventEmitter` (la classe `EventEmitter` du module `events`). La documentation d'une classe d'émetteurs commence toujours par la liste des types d'événements que ses instances peuvent émettre, *e.g.* `"connect"`, `"clientError"`, ... Les instances d'une classe héritière d'une classe d'émetteurs sont susceptibles d'émettre tous les types d'événements de sa classe parente.

On peut librement définir de nouveaux types d'événements et des écouteurs pour ces événements. La méthode `emit` d'un émetteur peut être invoquée pour émetre de manière sychrone un certain type d'événement vers ses écouteurs :

```
const EventEmitter = require('events');
let ee = new EventEmitter();
ee.on("test", v => console.log("ecouteur 1 : " + v));
ee.on("test", v => console.log("ecouteur 2 : " + v));
ee.emit("test", 42); // =>  ecouteur 1 : 42 // =>  ecouteur 2 : 42
```

3.2 Le cas des flux

Les flux dans Node sont des émetteurs dont les classes sont héritières de `stream.Readable` (flux en lecture) ou `stream.Writable` (flux en écriture), elles-mêmes héritières de la classe `events.EventEmitter`. Sans détailler le fonctionnement exact des flux, on peut mentionner les points suivants :

- La méthode `write` permet d'ajouter à un flux en écriture une nouvelle tranche de données (chunk) . La méthode `end` permet de signaler qu'un flux en écriture ne sera plus alimenté, en lui ajoutant de manière optionnelle une dernière tranche.
- Les données d'un flux en lecture ne peuvent être lues que s'il est muni d'au moins un écouteur d'événements de type `"readable"`, `"data"`, ou `"end"`.

– Un événement de type `"data"` est émis par un flux en lecture à chaque fois qu'il peut délivrer une nouvelle tranche de données. Un événement final `"end"` est émis lorsqu'il ne produira plus de nouvelles données,

4 Requêtes et réponses

Reprenons le code donné en exemple dans l'introduction. La partie `require` a déjà été expliquée plus haut. Le sens des invocations `createServer` et de `listen` est bien ce suggère leur nom :

1. La fonction de rappel de `createServer` sera celle exécutée à chaque nouvelle requête reçue par l'environnement sur l'un des ports d'écoute du serveur, (paramètre `req`), en fournissant aussi à cette fonction une réponse (paramètre `res`) librement initialisable (code de retour, headers) avant de spécifiant le corps de celle-ci (`end`).
2. La méthode `listen` de l'objet `server` permet d'effectuer une demande d'ouverture en écoute d'un port (`8080`) du serveur – la demande est asociée à un callback exécuté après cette ouverture.
3. Une fois la fin du code atteinte, le serveur entre dans état d'attente de nouvelles requêtes, état qui ne peut être interrompu que par une exception non capturée, une erreur, une fin d'exécution volontaire (`process.exit()`), ou encore un Ctrl-C (ou un `kill`) dans le terminal.

Il est important de comprendre que dans un appel de `createServer`, les objets `req` et `res` sont des *flux* : ce sont respectivement des instances des classes `http.IncomingMessage` et `http.ServerResponse`, héritieres respectivement des classes `stream.Reader` et `stream.Writer`, toutes deux héritières de `events.EventEmitter`.

4.1 L'objet `req`

Dans le corps de la fonction de rappel de `createServer`, l'objet `req` est un flux en lecture dont les données sont le corps de la requête reçue, mais à ce point de l'exécution, les seules informations fournies par `req` sont la méthode de cette requête et ses headers :

```
// dans le corps du callback de createServer :
console.log(req.method);   //=> POST
console.log(req.headers);  //=> { host: '127.0.0.1:8080', ... }
```

L'impossiblité de lire les données du flux `req` est simplement due au fait qu'il n'a pas encore d'écouteurs. Si l'on souhaite récupérer le corps de la requête, il faut ajouter à `req` des écouteurs d'événements en `"data"` et `"end"`[2], *c.f.* ci-dessus, pour lire ce corps de manière asynchrone et par tranches successives.

Ces événements ne seront émis par le flux `req` *qu'après* l'exécution complète de la fonction de rappel de `createServer`. En particulier, si la réponse du serveur dépend du contenu du corps la requête, la réponse ne pourra être spécifiée que dans un écouteur d'événements `"end"` de `req`. Voici un exemple d'ajout d'écouteurs, en supposant que le corps de la requête est un simple contenu textuel :

2. On peut aussi utiliser un unique écouteur en `"readable"`, *c.f.* la documentation de `stream.Reader`.

```
let req_body = "";         // accumulera les caractères du corps de req
req.setEncoding('utf-8'); // convertir chaque nouvelle tranche en chaîne.
req.on('data', chunk => { // à la disponibilité d'une nouvelle tranche,
  req_body += chunk;       // accumuler cette tranche.
});
req.on('end', () => {     // à la fin de la lecture du corps,
  // construire le corps de la réponse res à partir de corps de req
});
```

Par défaut, les tranches délivrées à l'écouteur sont des données numériques brutes encapsulées dans des objets de classe `Buffer` (*e.g.* `<Buffer 48 65 6c 6c 6f>`). L'invocation `req.setEncoding('utf-8')` force la conversion de chaque tranche délivrée en chaînes de caractères `utf-8` (*e.g.* `"Hello"`).

4.2 L'objet `res`

L'objet `res` est un flux en écriture muni de méthodes permettant de spécifier les headers de la réponse qui sera renvoyée par le serveur : `setHeader(name, value)`, `hasHeader(name)`, `getHeader(name)`), `removeHeader(name)` :

```
res.setHeader('Content-Type', 'text/plain; charset=UTF-8');
res.setHeader('Content-Encoding', 'gzip');
```

On peut remplir ce flux par les différentes tranches de la future réponse, à l'aide d'une suite d'invocations de la méthode `write` suivies d'une invocation finale de la méthode `end` :

```
res.write('Hello ');   // première tranche
res.write('World');    // tranche suivante
res.end('!');          // tranche finale
```

Ces invocations de `write` et `end` n'envoient pas de manière directe une réponse au client, elles ne font qu'alimenter en entrée le flux `res`. C'est seulement lorsque la fonction de rappel dans laquelle se trouve l'invocation finale `res.end()` – celle de `createServer`, ou encore un écouteur de `req` (*c.f.* ci-dessus), ou tout autre fonction de rappel voyant l'objet `req` – aura fini son exécution que l'environnement enverra de manière asynchrone les headers spécifiés par `res`, puis le corps de la réponse en délivrant les tranches accumulées par le flux `res`.

Remarque. Si la réponse contient un header `'Content-Length'` spécifiant une certaine longueur `length`, seuls les `length` premiers octets du corps de la réponse seront envoyés au client.

4.3 Un exemple d'échange client-serveur

Nous allons à présent tester un serveur en lui envoyant une requête depuis une autre instance de Node jouant le rôle de client, et en examinant les différentes étapes de leur échange. Après la commande `node server.js` dans un terminal, la commande `node client.js` sera lancée depuis un autre terminal.

```
// fichier server.js
const http = require('http');
const hostname = '127.0.0.1';
const port = 8080;
const server = http.createServer((req, res) => { // (2)
  let body = "";
  req.setEncoding('utf-8');
  req.on('data', chunk => { // (4) "Anyone" // (5) "There" //(6) "?"
    body += chunk;
  });
  req.on('end', () => { // (7)
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain; charset=UTF-8');
    res.write("Reply to: '" + body + "' ");
    res.write('Hello '); res.write('World'); res.end('!');
  });
  //(3)
});
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

```
// fichier client.js
const options = {
  hostname: '127.0.0.1',
  port: 8080,
  path: '/',
  method: 'POST',
  headers: { 'Content-Type': 'text/plain' }
};
const req = http.request(options, (res) => { // (8)
  let body = "";
  res.setEncoding('utf-8');
  res.on('data', (chunk) => {
    // (10) "Reply to:..." // (11) "Hello" // (12) "World" // (13) "!"
    body += chunk;
  });
  res.on('end', () => { // (14)
    console.log (body);
  });
  // (9)
});
// (1)
req.write("Anyone ");  req.write("there");  req.end("?");
```

La commande `node client.js` affichera dans le terminal du client le message suivant, avant de lui rendre la main : `In reply to: 'Anyone there?' Hello World!`. Les chiffres entre parenthèses indiquent ici l'ordre d'exécution.

- À l'exécution de `client.js`, l'invocation de `http.request` renvoie une instance de `http.ClientRequest`, une classe héritière de `stream.Writer`. En `(1)`, le flux `req` est alimenté en entrée par les différentes tranches de ce qui sera le corps de la requête HTTP envoyée au serveur.

  Lorsque s'achève l'exécution de la suite principale d'instructions de `client.js`, la requête HTTP spécifiée par `req` est effectivement envoyée au serveur, et l'environnement d'exécution se met dans un état d'attente de sa réponse.
- À la réception de cette requête, l'environnement d'exécution de `script.js` commence l'exécution de la fonction de rappel de `createServer` en `(2)`. Cette fonction termine son exécution en `(3)` après avoir ajouté des écouteurs au flux `req`. L'environnement commence alors l'émission d'événements correspondant à l'état courant du flux.
- L'écouteur en `'data'` de `req` est appelé trois fois de suite en `(4)`, `(5)`, `(6)`, une fois pour chaque tranche du corps de la requête. L'écouteur en `'end'` est enfin appelé en `(7)`. Il spécifie dans `res` les headers et les différentes tranches du corps de la réponse HTTP qui sera renvoyée au client.

  Lorsque s'achève l'exécution de l'écouteur, l'environnement d'éxécution émet effectivement la réponse HTTP spécifiée par `res` – à ce point, la requête a été entièrement gérée par le serveur, et l'environnement se remet dans un état d'attente de nouvelles requêtes.
- L'environnement d'exécution de `client.js` reçoit la réponse du serveur, construit un objet `res` de classe `http.IncomingMessage` à partir de cette réponse, puis invoque la fonction de rappel de `http.request` : l'exécution de `client.js` reprend en `(8)`. La fonction termine son exécution en `(9)`, après avoir ajouté des écouteurs au flux `res`. L'environnement commence alors l'émission d'événements correspondant à l'état courant du flux.
- L'écouteur en `'data'` de `res` est appelé quatre fois de suite en `(10)`, `(11)`, `(12)`, `(13)`, une fois pour chaque tranche du corps de la requête. L'écouteur en `'end'` est enfin appelé en `(14)` : à ce stade, le corps de la réponse à été entièrement accumulé et peut être affiché. L'exécution de la commande `node client.js` est à ce stade achevée, et le terminal reprend la main.

Cet exemple simpliste ne fait que montrer le principe d'un échange entre un client et un serveur Node. Nous verrons au chapitre suivant comment accéder à une base de données depuis le serveur ou encore comment extraire d'une requête les données d'un formulaire web, mais plutôt avec de nouveaux modules qui ne sont pas dans la distribution standard de Node[3].

3. La communication avec une base de données n'est pas gérée par les modules de cette distribution. Le module qui permettait d'extraire les données d'un formulaire pose des problèmes de sécurité, et ne doit plus être utilisé. Il est possible d'utiliser le constructeur `URL` de JavaScript pour effectuer une telle opération, mais aux prix de quelques contorsions.