Programmation Web Avancée

Cours 6

kn@lri.fr

université
PARIS-SACLAY

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓

2 Expressions régulières/Evènements/DOM ✓

3 Tableaux/JSON/AJAX/Asynchronisme ✓

4 Nodejs

- 4.1 Historique
- 4.2 Présentation
- 4.3 Modules en Nodejs
- 4.4 Modules en Javascript
- 4.5 Démo

Programmation Web

Côté client

- Javascript
- ~~ActionScript (Flash)~~
- ~~Applet Java~~

Côté serveur

- Java (JSP)
- PHP
- Python (Django)
- Ruby (Rails)
- OCaml (Ocsigen)

Problèmes d'avoir deux langages : pas de partage de code, pas de partage de compétences, infrastructure de tests séparées, différents paradigmes

Modèle classique de la programmation Web Client/Serveur

1. Client charge la page
2. Animations locales en JS
3. Remplissage de formulaire
4. Envoie d'une requête HTTP
5. Goto 1

Besoin d'interactions fines avec le serveur ⇒ **`XMLHttpRequest`**

Quel impact ?

Code serveur classique (PHP ou Java/JSP)

- Code séquentiel, procédural
- Pas prévu pour le cas « code appelé plusieurs fois de manière asynchrone »
- Pas de modèle de gestion de concurrence (les exécutions concurrentes sont isolées les unes des autres), chacune dans un processus ou thread

Exemple : dépôt de fichier et bare de chargement côté client

- Beaucoup de petites requêtes HTTP (le client « ping » le serveur qui répond avec la taille courante du fichier uploadé)
- Chaque requête est traitée par un fork ou un thread

⇒ problème de performances

Constat fait par Ryan Dahl en 2009 ⇒ création de Nodejs

Principes de Nodejs

- Machine virtuelle Javascript basé sur V8 (google chrome)
- Support pour Javascript 6 et suivants
- Orienté programmation Web côté serveur
- Basé sur de la programmation *asynchrone* et une boucle d'évènements
- Un (ou plusieurs 😬) système de modules
- Un système de gestion de packages (`npm`)

Avantages :

- Javacript est un langage connu
- Le modèle gestion d'évènement est bien compris, simple
- Allocation fine des ressources :
  - La boucle d'interaction est en attente d'évènements (requête HTTP, I/O, ...)
  - Déclanche un callback
  - Pas de synchronisation, modèle mono-thread
  - Worker si besoin de faire du multi-thread

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓

2 Expressions régulières/Evènements/DOM ✓

3 Tableaux/JSON/AJAX/Asynchronisme ✓

4 Nodejs

- 4.1 Historique ✓
- 4.2 Présentation
- 4.3 Modules en Nodejs
- 4.4 Modules en Javascript
- 4.5 Démo

L'interpréteur nodejs (console)


```
$ node
> console.log("Hello, world !");
Hello, world!
undefined
> let answer = 42;
undefined
> 1 + answer;
43
```

Fonctionne comme la console d'un navigateur, affiche le résultat de l'expression

Possibilité de compléter avec [TAB]

L'interpréteur nodejs (script)


```
//Fichier hello.js
function helloWorld() {
  console.log("Hello, World!");
}

helloWorld();
```

Dans un terminal :

```
$ node hello.js
Hello, World!
```

Différence entre la console et le mode programme

- En console, tous les modules sont ouverts par défaut
- En console, **`this`** est associé à l'objet global
- Un fichier représente un *module*, aucun module n'est ouvert par défaut
- Dans un module, **`this`** est associé au module courant

⇒ Un symbole global (fonction, variable, constante, classe, ...) n'est pas visible par défaut dans les autres fichiers.

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓

2 Expressions régulières/Evènements/DOM ✓

3 Tableaux/JSON/AJAX/Asynchronisme ✓

4 Nodejs

- 4.1 Historique ✓
- 4.2 Présentation ✓
- 4.3 Modules en Nodejs
- 4.4 Modules en Javascript
- 4.5 Démo

Qu'est-ce qu'un module ? (en général)

Une portion *close* de code regroupant des concepts (fonction, type, constante) reliés.
Un module fournit :

- Un ensemble de symboles exportés
- Un ensemble de symboles privés
- La liste des modules dont il dépend

A priori, un module *n'est pas extensible* et n'a pas de dépendences cycliques.
Quelques langages avec modules et sans:

- Avec modules: Python, Ruby, OCaml, C#, Java (depuis 9.0), Rust, Go, Javascript,...
- Sans modules: Java (avant 9.0), C, C++, ...

Choses qui ne sont pas des modules

Classes (au sens Java ou C++) :

- Ce sont des types
- Nécessitent de créer des instances (objets)
- Peuvent êtres étendues (héritage)
- Dépendences cycliques autorisées (mais pas pour l'héritage)

Namespaces (au sens C++) :

- Résout uniquement le problème de noms
- Peut être étendu
- Objets syntaxiques

Headers (.h de C ou C++)

- Purement syntaxique (inclusion de fichier)

Définition d'un module (format CommonJS)

Au sein d'un fichier ***hello.js*** :

```
const message = "Hello, world!";
function print(s) {
  console.log (s);
}
function helloWorld() {
  print (message)
}
module.exports.message = message;
module.exports.helloWorld = helloWorld;
```

Dans le fichier *main.js*

```
const hello = require ("./hello.js");

hello.helloWorld();
console.log(hello.message.length);
```

`hello.print` n'est pas visible à l'extérieur.

Sémantique des modules nodejs (simplifiée)

- La fonction prédéfinie `require (…)` charge le fichier dont le nom est donné en argument.
- Le code présent dans le fichier est évalué. Un objet global `module` est défini au début de l'évaluation du code
- Il contient une propriété `exports` contenant un objet vide
- À la fin de l'évalution, la fonction `require(…)` renvoie l'objet que le fichier a stocké dans `module.exports`

Avantages :

- Sémantique simple
- Utilise les concepts de portées et d'objets présents en JS
- Le langage permet de nombreuses fonctionalité : import partiel, renommage, ...

Exemple d'utilisation avancée

```
//fichier arith.js
function f (x) { … }
function g (x, y) { … }
function h (x, y) { … }

module.exports = { neg : f, add : g, mult : h };
-----------------------------------------------

//fichier user.js
const math = require ('./arith.js');
// on peut utiliser math.neg, math.add, math.mult

const { neg, mult } = require ('./arith.js');
// on peut utiliser la notation générique pour
// associer directement aux variables neg et mult les champs
// neg et mult de l'objet renvoyé par require
// ici on peut utiliser neg/mult sans préfixe
```

Modules de la bibliothèque standard

Pour ces derniers, on ne met pas de chemin ni d'extension

L'usage veut que l'on définisse une variable qui s'appelle comme le module

```
const fs = require ('fs');
const http = require ('http');
```

Il est aussi plus propre de définir la variable contenant le module avec **`const`**, pour empêcher une modification non voulue.

```
let fs = require ('fs');
const http = require ('http');

fs = 42;  //ooops, fs ne contient plus un module mais un Number
http = 42; //erreur, car http est const.
```

Quelques modules utiles

`child_process` : Lancement de programmes (exec, fork, ...)
`console` : Console de débuggage (console.log, console.debug, ...). Ouvert automatiquement, pas besoin de **`require`**.
`dns` : résolution de noms de domaines
`fs` : Système de fichier (ouverture de fichier, répertoires, ...)
`http` : Serveur HTTP simple.
`https` : Serveur HTTPS simple.
`os` : Informations sur le système (nombre de CPUs, OS, ...)
`process` : Gestion des processus (envoie de messages, kill, ...)
`querystring` : décodage/encodage des chaînes de la forme: **`?a=b&c=%20&…`**
`timer` : Expose les fonctions **`setTimeout`** et **`setInterval`**. Ouvert automatiquement.
`zlib` : Lecture/écriture d'archives zip

Bien d'autres modules utilitaires. Pas beaucoup de structures de données.

Synchrone/asynchrone

La plupart des fonctions « lentes » (i.e. potentiellement soumise à l'attente d'un évènement externe) sont *asynchrones*. Elle prennent en argument supplémentaire un **`callback`** appelé sur le résultat de la fonction.

Souvent (mais pas tout le temps) des version synchrones existent. Leur nom contient le suffixe **Sync**.

```
const fs = require ('fs');

fs.copyFile("fichier.txt", "copie.txt", (err) => {
  if (err) throw err; //leve une exception si err est non null
  console.log("Copie terminée");
});


fs.copyFileSync("fichier.txt", "copie.txt");
//leve une exception en cas d'erreur
console.log("Copie terminée");
```

Quelle est la différence entre ces deux versions ?

Synchrone/asynchrone

La version *asynchrone* rend la main au runtime nodejs. Pas d'attente active, d'autres évènements peuvent êtres déclanchés pendant la copie.

La version synchrone bloque tout le runtime, jusqu'à la fin de la copie.

 La bibliothèque standard nodejs n'utilisait pas de promesse jusqu'à la version 11.x

C'est maintenant possible, mais les fonctions qui renvoient les promesses sont dans des sous-modules particuliers :

```
const fs = require ('fs/promises');

await fs.copyFile("fichier.txt", "copie.txt");
//lève une exception en cas d'erreur
console.log("Copie terminée");
```

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓

2 Expressions régulières/Evènements/DOM ✓

3 Tableaux/JSON/AJAX/Asynchronisme ✓

4 Nodejs

- 4.1 Historique ✓
- 4.2 Présentation ✓
- 4.3 Modules en Nodejs ✓
- 4.4 Modules en Javascript
- 4.5 Démo

Histoire des modules JS

- 1995-2009 : le Moyen Âge (incompatibilités, pas de standard, IE, ...)
- 2008 : Google Chrome™ (javascript *performant*, ...)
- 2009 : EcmaScript 5.1 (mode strict, ...). Le Web *côté client* se met en place
- 2009 : Premier prototype de NodeJS

Programmes écrit en JS côté serveur ⇒ nécessité d'un système de modules (CommonJS, dont celui de NodeJS est un successeur)

En parallèle, proposition d'autres types de modules adapté au JS dans le navigateur

2015 : EcmaScript 6, proposition d'un système de modules dans le standard

Problème : fait pour concilier JS Client et JS serveur, pas encore supporté ni finalisé

Exemple de modules ES


```
//fichier mod.js
import * as fs from 'fs'
import 'other_module'

fs.copyFile(…);

f (…);  // définit comme defaut dans 'other_module'

export function publicFun(…) { … }
export default function foo() { … }
```

Support préliminaire dans NodeJS 8.x avec **`--experimental-modules`**

Support stable depuis NodeJS 13.x

Support dans les navigateurs depuis 2018

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓

2 Expressions régulières/Evènements/DOM ✓

3 Tableaux/JSON/AJAX/Asynchronisme ✓

4 Nodejs

- 4.1 Historique ✓
- 4.2 Présentation ✓
- 4.3 Modules en Nodejs ✓
- 4.4 Modules en Javascript ✓
- 4.5 Démo

Démo : serveur Web

On veut créer un petit serveur Web minimal :

- Maintient un compteur global
- Répond à chaque requête HTTP avec un JSON contenant le compteur et un timestamp
- Incrémente le compteur
- Si un paramètre **`reset=true`** est passé, on réinitialise le compteur.

Nodejs ...

- Du javascript standard et efficace
- Modèle évènementiel simple, monothread
- Système de module simple et robuste

Il manque encore des morceaux :

- Gestion de packages
- Système de build
- Framework Web (on ne va pas décoder et traiter les requêtes HTTP manuellement)