

PW6 — Programmation Web

Peter Habermehl, Vincent Padovani, Aldric Degorre

L3 Informatique générale — Université de Paris

(Adapté d'après transparents de Cristina Sarangelo.)

Développement Web côté serveur avec node.js et express.js

- <https://nodejs.org>
- <https://expressjs.com>
- <https://www.npmjs.com>

Node.js

Qu'est-ce que c'est ?

- Node.js est un outil *open-source* pour le développement d'applications Javascript côté serveur
- Il contient :
 - un moteur Javascript (le même que celui utilisé par Google Chrome)
 - une API publique (sous forme de plusieurs modules) -- appelée *node core* -- pour accéder à une variété de ressources (système de fichiers, réseau, etc.) avec Javascript
 - un outil en ligne de commande

Développer une application Javascript côté serveur avec node.js

- Node.js sera installé sur la machine sur laquelle le serveur doit tourner
- pour installer node.js sur votre machine : <http://nodejs.org/download/>
 - (déjà installé sur les machines de l'UFR)
- Après installation, l'outil en ligne de commande node sera disponible.

- Avec node on peut exécuter du code Javascript qui utilise les modules installés par node.js, ainsi que d'autres modules qu'on peut explicitement installer
- Pour implementer un serveur :
 - L'écrire en Javascript en incluant les modules node.js dont on a besoin
 - Le lancer dans node : aller dans le répertoire qui contient `mon_serveur.js` :

```
$ node mon_serveur.js
```

Installer d'autres modules depuis npm

- npm (*node packaged modules*) : une très large collection de modules pour node.js
- En ligne commande, on peut installer des nouveaux modules depuis npm :

```
$ npm install nom_du_module      # installation locale  
$ npm install nom_du_module -g  # installation globale
```

- Le plus souvent on exécutera la commande ci-dessus depuis le répertoire contenant le code Javascript.

Installation locale

1. cherche le répertoire `node_modules` le plus proche (en remontant du répertoire courant vers la racine), s'il n'existe pas il le crée dans le répertoire courant.
2. Installe le module dans `node_modules/nom_du_module`

Installation globale (pas autorisée sur les machines de l'UFR)

- installe le module demandé dans un sous-répertoire `nom_du_module`
- d'un répertoire `node_modules` pre-défini
 - (`/usr/local/lib/node_modules` typiquement)
 - de plus installe la commande `nom_du_module`

Désinstaller un module local

dans le répertoire de `npm install` :

```
$ npm uninstall nom_du_module
```

Désinstaller un module global

```
$ npm uninstall -g nom_du_module
```


Modules node.js

- Le core de node.js offre plusieurs modules dont :
 - `fs` : pour travailler avec le système de fichiers
 - `http` : pour gérer le protocole http
 - `net` , `udp` : pour opérer à travers le réseau
 - ...(moins d'une trentaine au total)
- Des dizaines de milliers de modules disponibles sur npm !
- Le module le plus utilisé pour développer des serveurs Web est `express`

```
$ npm install express
```

Utilisation des modules en node.js

- Pour utiliser un module `nom_du_module` dans le code Javascript utiliser l'instruction :

```
var m = require('nom_du_module');
```

- `require` renvoie un objet Javascript
- On pourra ensuite utiliser sur `m` toutes les méthodes exportées par le module `nom_du_module`
 - Chaque module offre son ensemble de méthodes

Exemples de méthodes

- Dans `fs` :

```
var fs = require('fs');  
fs.readFile(...)  
fs.writeFile(...)  
fs.mkdir(...)  
fs.rename(...)  
...
```

- Dans `http` :

```
var http = require('http');  
http.createServer(...); //crée un serveur qui peut  
répondre  
// à des requêtes HTTP  
http.request(...) // envoie une requête HTTP à un serveur  
...
```

Méthodes asynchrones

Beaucoup de méthodes node.js sont *asynchrones* et répondent à un principe de *programmation événementielle*.

Exemple

- Dans l'interaction avec le système de fichiers, via le module `fs` :

```
var fs = require('fs');
fs.readFile('/etc/passwd', function (erreur, donnees) {
  if (erreur) throw erreur;
  console.log(donnees.toString('utf8'));
});
console.log('en attendant la lecture du fichier...');
// instructions
```

- Dans l'interaction avec le système de fichiers, via le module `fs` :

```
var fs = require('fs');
fs.readFile('/etc/passwd', function (erreur, donnees) {
  if (erreur) throw erreur;
  console.log(donnees.toString('utf8'));
});
console.log('en attendant la lecture du fichier...');
// instructions
```

- l'appel à `readFile` démarre la lecture du contenu du fichier et associe la fonction passée en argument comme listener de l'événement « lecture terminée »
- à la lecture terminée, la fonction (de *callback*) sera exécutée avec les arguments :
 - l'éventuel erreur produit dans la lecture (premier paramètre)
 - le contenu du fichier (deuxième paramètre)

```
var fs = require('fs');
fs.readFile('/etc/passwd', function (erreur, donnees) {
  if (erreur) throw erreur;
  console.log(donnees.toString('utf8'));
});
console.log('en attendant la lecture du fichier...');
// instructions
```

- La méthode `readFile` est **non bloquante** : elle n'attend pas que la lecture du fichier soit terminée et la fonction de *callback* exécutée.
- Conséquence : les instructions qui suivent `fs.readFile` seront exécutées **avant** la fonction de *callback* (sans attendre la fin d'une lecture potentiellement lente).

- Si on veut imposer qu'un bloc d'instructions soit exécuté seulement à lecture terminée, il faut inclure ces instructions dans la fonction de *callback*

```
var fs = require('fs');
fs.readFile('/etc/passwd', function (erreur, donnees) {
  if (erreur) throw erreur;
  console.log(donnees.toString('utf8'));
  // instructions <-- ici !
});
console.log('en attendant la lecture du fichier...');
```

Introduction au module http

```
var http = require('http');
```

Méthode principale :

```
var serv = http.createServer(fonction);
```

- Retourne un objet de la classe `http.Server` qui émet des événements liés au protocole HTTP.
- Événement principal : « *request* », émis à chaque fois que le serveur reçoit une requête HTTP
- La fonction passée à la création du serveur est invoquée à chaque événement de type « *request* »
- Pour que le serveur créé commence à accepter des connexions sur un port :
`serv.listen(port)`

Requête et réponse

- La fonction qui gère l'événement de "request" reçoit deux arguments :

```
var serv = http.createServer(function(request, response){  
    ...  
});
```

- `request` : la requête HTTP reçue
 - `response` : la réponse HTTP à envoyer
- Plusieurs méthodes sont disponibles sur les objets `request` et `response`
- En particulier ces méthodes permettent d'envoyer la réponse en plusieurs fois

Envoyer la réponse HTTP

- `response.writeHead(statusCode[, headers])` envoie l'entête de la réponse HTTP. Exemple :

```
var body = 'hello world';
res.writeHead(200, {
  'Content-Length': body.length,
  'Content-Type': 'text/plain' });
```

- status code : `200` (success), `404` (not found), ...
- `writeHead` peut être appelé une seule fois et avant de terminer la réponse
- si une partie du body est envoyée — ou la réponse terminée — avant d'appeler `writeHead`, un entête par défaut est calculé et envoyé
- pas obligatoire de spécifier tous les en-tête (*content-length*, *content-type*, *connection*, *host*, *accept*, etc.)
 - les en-têtes pas spécifiés prendront une valeur par défaut

- `response.write(string)` : envoie un fragment du *body* de la réponse HTTP
 - `write` peut être appelé plusieurs fois pour envoyer la réponse en plusieurs morceaux
- `response.end()` : termine la réponse HTTP
 - doit être appelé sur chaque réponse

Introduction au module `express.js`

`express.js`

- est un module node.js (`express`) pour le développement d'applications Web et mobile (*web framework*)
- permet une gestion plus haut-niveau du cycle requête-réponse HTTP par rapport au module `http`

Utilisation d'express.js

- Inclusion du module :

```
var express = require('express');
```

- création d'un objet express (le serveur) :

```
var serv= express();
```

- "mise en ligne" du serveur :

```
serv.listen(port);
```

- Mécanisme principal pour le développement du serveur : **définition de routes**
 - **route** : association d'un *handler* à un certain type de requête HTTP

Routes en express

- Une route `serv.METHOD(uri, fonction)` associe le *handler* `fonction` à l'événement suivant :
requête HTTP
 - avec méthode `METHOD` (e.g. `GET`, `POST`, ...)
 - vers l'URI `uri` (e.g. `'/about'`, `'/'`, `'/cours/td'`)
- `fonction` est appelée à chaque requête conforme à la route
- `fonction` reçoit deux arguments, typiquement appelés `req` et `res`
 - `req` : la requête HTTP
 - `res` : la réponse à envoyer

Exemples de routes GET

```
serv.get('/', function (req, res) {  
    res.send('requête de GET vers la homepage');  
});  
serv.get('/process', function (req, res) {  
    res.send('requête de GET vers /process');  
});
```

Si `serv` écoute sur le port 8080,

- la première fonction sera appelée à chaque fois qu'on se rend à l'adresse <http://localhost:8080/>
- la deuxième fonction à chaque fois qu'on se rend à l'adresse <http://localhost:8080/process>

Exemple de route POST

```
serv.post('/about', function (req, res) {  
  res.send('requête de POST vers /about');  
});
```

Si `serv` écoute sur le port 8080, la fonction sera exécutée à chaque fois qu'on soumet par exemple un formulaire du type :

```
<form method="post" action="http://localhost:8080/about">
```

Pour aller plus loin

- Pour plus de détails sur le routing : <http://expressjs.com/en/guide/routing.html>
- Des propriétés et méthodes sont disponibles pour manipuler `req` et `res`

Quelques propriétés utiles de la requête HTTP en express

- `req.originalUrl` : l'url de la requête (n'inclut pas le hostname). Ex. :

```
// GET /search?q=something  
req.originalUrl // => "/search?q=something"
```

- `req.hostname` : le host demandé par la requête (comme dans l'en-tête HTTP)

```
// Host: "example.com:3000"  
req.hostname // => "example.com"
```

- `req.query` contient les paramètres de la requête (utile pour le traitement des requêtes GET).

Exemple : requête GET dont l'URI se termine par

`/shoes?order=desc&shoe[color]=blue&shoe[type]=hogan`

- `req.query.order // => "desc"`
- `req.query.shoe.color // => "blue"`
- `req.query.shoe.type // => "hogan"`

- `req.body` contient les paramètres de la requête POST, disponible uniquement si on a ajouté un *middleware* du module `bodyParser` au serveur :

```
var bodyParser = require('body-parser');  
serv.use(bodyParser.urlencoded({ extended: false }));
```

(après l'avoir installé : `$ npm install body-parser`)

Exemples :

- POST avec paramètres

```
user[name]=tobi&user[email]=tobi@learnboost.com
```

- `req.body.user.name` => `"tobi"`
- `req.body.user.email` => `"tobi@learnboost.com"`

- POST avec paramètre `{ "name": "tobi" }`

- `req.body.name` => `"tobi"`

- Une route peut être associée à un URI contenant une partie variable

```
serv.get('/user/:nom', f );
```

- La valeur de `nom` est disponible dans la variable `req.params.nom` :

```
serv.get('/user/:nom', function (req, res) {  
    res.send(req.params.nom);  
});
```

```
// GET /user/cristina  
req.params.nom // => "cristina"
```

Envoyer une réponse HTTP avec express

- `res.send(data)` envoie la réponse HTTP avec pour contenu `data`
 - `data` peut être une chaîne de caractères, un objet ou un tableau
 - une façon d'envoyer du petit contenu HTML :

```
res.send(  
  '<!DOCTYPE html> <html><body>  
  <p> bienvenue sur ma page </p>  
  </body></html>');
```

- `res.end()` termine la réponse HTTP sans envoyer de données
 - Ex. `res.status(403).end(); //accès interdit` , termine la réponse
 - `res.status(status-code)` : modifie le status code de la réponse HTTP
 - Ex : `res.status(404); //not found`

- `res.download('chemin/fichier')` envoie le fichier spécifié, le navigateur proposera son téléchargement
- `res.render(fichier, objet)` traduit le contenu du fichier en HTML en invoquant un « *view engine* »
 - qui doit être explicitement ajouté au serveur, cf. plus loin)
 - envoie l'HTML résultant au client et termine la réponse HTTP
- et d'autre méthodes (cf. <http://expressjs.com/en/4x/api.html>)
- Remarque : comme `end()`, aussi `render()`, `send()` et `download()` (entre autres) envoient une réponse HTTP et terminent le cycle requête-réponse HTTP (pas besoin de `end()` explicite ensuite).

Utiliser le middleware en express

- les fonctions utilisées comme handlers pour les routes sont aussi appelées *middleware*. On peut...
 - en spécifier plusieurs pour gérer le même type de requête et le même uri:

```
serv.get('/', fonction1);  
serv.get('/', fonction2);
```

- également en associer plusieurs dans la même route :

```
serv.get('/', fonction3, fonction4, fonction5, ...);
```

- associer du middleware à toutes les requêtes (i.e. toute méthode, tout uri)

```
serv.use(fonction6, fonction7,...);
```

- ou à toutes les requêtes vers un certain uri (toute méthode)

```
serv.use('/', fonction6, fonction7,...);
```


- À la réception d'une requête pour `serv`, tout le *middleware* applicable à la requête ira dans une pile d'exécution
- Exemple, soit le middleware suivant monté sur `serv` :

```
serv.get('/about', fonction1);  
serv.get('/', fonction2, fonction3);  
serv.post('/', fonction4, fonction5);  
serv.use(fonction6, fonction7);  
serv.use('/about', fonction8);
```

- Une requête de GET pour la racine `'/'` aura la pile d'exécution suivante
fonction2
fonction3
fonction6
fonction7 ← tête de la pile
- Remarque : l'ordre dans la pile respecte l'ordre de définition du *middleware*

Pile d'exécution d'une requête HTTP

- La fonction à la tête de la pile est exécutée automatiquement
- Les autres sont exécutées uniquement si invoquées explicitement par la fonction précédente
- À cet effet chaque fonction de *middleware* dispose d'un argument en plus (en plus de `req` et `res`) : une référence à la prochaine fonction dans la pile d'exécution

```
serv.get('/about', function (req, res, next){  
    ...  
    next();  
});
```

- Si une fonction de *middleware* est exécutée et n'appelle pas la suivante avec `next()`, aucune des fonctions suivantes dans la pile sera exécutée

Gestion de la pile d'exécution:

Un *middleware* qui n'invoque pas la fonction suivante doit terminer la réponse HTTP

- avec une des méthodes disponibles : `send()` , `render()` , `end()` , etc.

Exemple

```
serv.use('/user/:id', function (req, res, next) {  
  if (req.params.id == 0) res.send ('OK');  
  else next();  
});  
serv.get('/user/:id', function (req, res, next) {  
  res.status(404).end();  
});
```

Si ce n'est pas le cas, la requête HTTP restera "pending"

Un *middleware* monté avec une route (i.e avec `serv.method`) peut également utiliser

```
next('route')
serv.get('/', function(req, res, next) {
    console.log('homepage demandée'); next('route')
},
function(req, res, next) {
    res.send('cette fonction n'est pas exécutée'); next()
});
serv.use(function (req, res) {
    res.send('cette fonction est exécutée');
});
```

Effet de `next('route')` : exécuter la prochaine fonction dans la pile après la "sous-pile" de la route courante

Pour plus de détails :

<http://expressjs.com/en/guide/using-middleware.html>

Embedded Javascript

- Un serveur express peut envoyer du HTML en réponse à une requête HTTP, comme argument de `res.send()`.
- L'HTML envoyé peut ainsi être dynamique :

```
res.send('<!DOCTYPE html> <html><body>  
  <p> bienvenue sur la page de ' + v_nom + '</p>  
  </body></html>');
```

- Toutefois cette solution est lourde si l'HTML est volumineux.
- On aimerait disposer de l'HTML dans un fichier à part, mais il faut une solution pour que l'HTML puisse contenir du code qui le rend dynamique.
- ***Embedded Javascript*** est un module node.js (appelé `ejs.js`) qui permet d'inclure et interpréter du Javascript dans un fichier .html
- Des tels documents HTML sont appelé *fichiers template*.

- Exemple de document HTML avec *embedded Javascript* `mapage.ejs` :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8">
    <title> Une page EJS </title>
  </head>
  <body>
    <h1> bienvenue sur la page de <%= v_nom %> </h1>
  </body>
</html>
```

- Un serveur express peut invoquer ejs pour interpréter un template .ejs et ainsi produire du HTML pur, avant de l'envoyer avec la réponse HTTP
- Cela demande de mettre en place un *view engine* (aussi dit *template engine*) pour le serveur express
- Associer `ejs` en tant que view engine au serveur express `serv` :

```
serv.set('view engine', 'ejs');
```

- Le module `ejs.js` doit être d'abord installé depuis npm
 - Pas besoin de `require('ejs')` : express le demandera implicitement.

- Après avoir mis en place `ejs` comme *view engine*, l'objet `res` peut envoyer des fichiers `.ejs` avec `res.render` :

```
res.render('mapage.ejs', {v_nom : 'cristina'});
```

- Cette instruction
 - invoque implicitement le *view engine* `ejs` qui interprète le javascript dans `mapage.ejs` en utilisant les valeurs des paramètres passées en deuxième argument
 - envoie l'html résultant et termine la réponse HTTP.
- Le deuxième argument de `res.render` est un objet Javascript, contenant un couple `param: valeur` pour chaque paramètre utilisé dans le *template*
- Attention : `ejs` cherche `mapage.ejs` dans un sous-répertoire appelé `views` du répertoire courant.

- Une simple extension de la syntaxe HTML, l'extension du fichier doit être `.ejs`
- Pour inclure du Javascript dans un document HTML utiliser la syntaxe :

```
<% du code javascript %>
```

- Pour produire une valeur dans le HTML :

```
<%= expression javascript %>
```

Exemples

```
<h1> bienvenue sur la page de ' <%= v_nom %> </h1>
```

```
<% if ( user.name != 'cristina' ) { %>  
    <h2> <%= user.name %> </h2>  
<% } %>
```

Attention : ne pas oublier « { » avant d'interrompre une instruction pour passer à HTML.

- Exemple avec boucle :

```
<ul>
  <% var attr;
  for (attr in user){ %>
    <li> <%= user[attr] %> </li>
  <% } %>
</ul>
```

Alors, avec `user == { prenom: 'Jean', nom: 'Dupond', age: 52 }`, `ejs` génère :

```
<ul>
  <li> Jean </li>
  <li> Dupond </li>
  <li> 52 </li>
</ul>
```

- Inclusion d'autres *templates* dans le *template* courant

```
<%- include chemin/fichier.ejs %>
```

- Le chemin est relatif au répertoire du fichier courant
- `include` peut avoir un deuxième argument : un objet contenant des paramètres à passer à `fichier.ejs` :

```
<%- include( 'chemin/fichier.ejs',  
{param1 : valeur, param2 : valeur, ... }) %>
```

- Pour plus de documentation sur `ejs` :
 - documentation incluse avec l'installation de `ejs.js` (`readme.md`)
 - un tutoriel utile :
<https://scotch.io/tutorials/use-ejs-to-template-your-node-application>

Rattacher des fichiers statiques à un template

- Un template (ejs) peut être attaché à des fichiers statiques (images, css, js coté client)
- Ces fichiers doivent se trouver dans un repertoire rendu accessible au serveur express. Soit `public` ce repertoire (on peut donner un autre nom)
- Arborescence possible :

```
|  
|-public  
|  |-monScript.js  
|  |-monStyle.css  
|  |-monImage.jpg  
|  
|-views  
|  |-maPage.ejs  
|  
|-monApp.js
```

- typiquement : `public` est placé dans le répertoire principal du projet

Pour rendre le repertoire accessible par le serveur express :

```
//monApp.js
var express = require('express');
var serv= express();
...
serv.use(
  express.static('public'));
...
serv.listen(8080);
```

Ensuite tout fichier dans `public` sera associé à l'url
<http://localhost:8080/fichier>

Donc dans les templates :

maPage.ejs :

```
<!DOCTYPE html>
<html>
  <head> ...
    <link href="/monStyle.css" rel="stylesheet">
    <script src="/monScript.js"></script>
  </head>
  <body>
    ...
    <img src= "/monImage.jpg">
    ...
  </body>
</html>
```

ne pas oublier « / » !

On peut spécifier plusieurs répertoires statiques dans le serveur express. Cela permet par exemple d'organiser les fichiers statiques en sous-répertoires

```
|  
|-public  
|  |-js  
|    |-monScript.js ...  
|  |-css  
|    |-monStyle.css  
|  |-images  
|    |-monImage.jpg  
|-views  
  |-maPage.ejs  
  |-monApp.js
```


monApp.js

```
var express = require('express');  
var serv= express();  
...  
serv.use(express.static('public/images'));  
serv.use(express.static('public/css'));  
serv.use(express.static('public/js'));  
...  
serv.listen(8080);
```

=> fichiers toujours associés aux url :

localhost:8080/monScript.js

localhost:8080/monStyle.css

localhost:8080/monImage.css

Connexion à une base de données

- Un serveur express peut se connecter à une base de données et la manipuler
- La connexion à la base est gérée entièrement par un autre module node.js, indépendant de express
- npm offre un module différent pour chaque SGBD majeur
- pour installer `mysql` :

```
npm install mysql
```

Utiliser le module mysql

- Inclusion du module :

```
var mysql = require('mysql');
```

- Création d'un objet connexion :

```
var connection = mysql.createConnection({  
    host : 'localhost',  
    user : 'username',  
    password : 'pwd',  
    database : 'db_name'  
});
```

- La fonction de *callback* pour `connection.query` reçoit trois paramètres:
 - `err` : éventuel erreur d'exécution
 - `rows` : le résultat de la requête sous forme d'un tableau d'objets
 - chaque ligne `rows[i]` est un objet de la forme :
`{champ1 : 'valeur1', ... , champ n : 'valeurn'}`
 - `fields` : un tableau contenant un objet pour chaque attribut du résultat
 - `fields[i].name` renvoie le nom de l'attribut `i` du résultat
- Pour plus de détails : <https://www.npmjs.com/package/mysql>

Créer un module node.js

Pour rendre le code node.js modulaire on peut créer des modules

- Il suffit d'exporter des fonctions ou objets. Dans `monModule.js` :

```
module.exports= function(a, b) {...}
```

- Ensuite importer `monModule.js`, comme n'importe quel autre module, pour utiliser la fonction exportée. Dans `monApp.js` :

```
var f = require('./monModule');  
var z = f(2,3);
```

```
|  
|-monModule.js  
|-monApp.js
```

- On peut également exporter un objet.

- Dans `monModule.js` :

```
module.exports= {  
  prenom: "jean" , nom: "dupond",  
  nomComple : function() {  
    return this.prenom+' '+ this.nom }  
}
```

- Dans `monApp.js` :

```
var o = require('./monModule');  
var z = o.nomComple()
```

- Un répertoire `monModule` contenant un `index.js` peut remplacer `monModule.js`

```
| -monModule  
|   |-index.js  
|  
| -monApp.js
```

Créer un module node.js : le fichier package.json

- On peut ajouter des meta-données à son module
- Cela permet par exemple de changer le nom/parcours du fichier principal
- Créer un répertoire `monModule` contenant
- un fichier `mon_code.js` contenant le code à exporter (l'ancien `index.js`)
- un fichier `package.json` contenant des meta-données dans ce format :

```
{  
  "name" : "monModule",  
  "version" : "1.0.0",  
  "main" : "mon_code.js"  
}
```

- package.json peut également lister les modules dont notre module dépend :

```
{  
  "name" : "monModule",  
  "version" : "1.0.0",  
  "main" : "mon_code.js",  
  "dependencies" : {  
    "ejs": ">=2.3.3",  
    "express": "4.x"  
  }  
}
```


- `$ npm install` exécuté depuis le répertoire `monModule`
 - cherche un fichier `package.json` dans le répertoire courant.
 - s'il est trouvé, tous les modules listés dans `dependencies` installés, dans la version indiquée
- Pour installer un nouveau module npm et automatiquement le lister dans `dependencies` , exécuter depuis le répertoire `monModule` :

```
$ npm install nouveau_module --save`
```

- Pour plus de détails : <https://docs.npmjs.com/files/package.json>

Créer un module Router avec express

- express a récemment introduit la classe Router qui permet de
 - encapsuler un ensemble de routes "relatives"
 - les exporter dans leur ensemble
 - les attacher à un autre serveur express en le montant sur un chemin racine

(cf. prochain transparent)

```
|  
|-monRouter.js  
|-monApp.js
```

- Dans `monRouter.js`

```
var express = require('express');  
var router = express.Router();  
//on attache des routes à router de la même façon que à un serveur express()  
router.get('/', fonction2);  
router.post('/check', fonction3);  
module.exports = router;
```

- Dans `monApp.js`

```
var express = require('express'); var serv= express();  
var route = require ('./monRouter');  
serv.use('/register', route);...  
serv.listen(8080);
```

Les routes GET `/register` et POST `/register/check` seront gérées par `route`.

Comment fonctionne `require()`

Deux formes

1. `require('relative or absolute path/nom_du_module')`
`relative or absolute path` : un chemin qui commence par `'/'`, `'../'`, ou `'./'`.

Dans le répertoire spécifié par `relative or absolute path`, cherche soit `nom_du_module.js` soit `nom_du_module/index.js`
(`index` ou autre nom spécifié dans `nom_du_module/package.json`)

2. `require('nom_du_module')`

- cherche d'abord un module du *node core* de nom `nom_du_module`
- s'il n'est pas trouvé, cherche (à commencer par le répertoire courant, puis en remontant vers la racine) un répertoire appelé `node_modules` et, dans ce répertoire, cherche soit `nom_du_module.js` soit `nom_du_module/index.js` (`index` ou autre nom spécifié dans `nom_du_module/package.json`)
- Si le module n'a pas encore été trouvé, répète la recherche dans des répertoires `node_modules` pre-définis.