# Programmation web
# JavaScript - Applications 3 - Les packages de Node

Vincent Padovani, PPS, IRIF

Ce chapitre présente trois des multiples *packages* disponibles pour Node, des collections de modules supplémentaires pour sa version de base.

## 1 Installations de packages supplémentaires

L'installation de nouveau groupes de modules pour Node se fait à l'aide du "node package manager", invocable par la commande `npm` (qui est intégrée à la distribution de Node). Les packages peuvent être installés de deux manières :

1. **Localement.** Pour installer par exemple comme premiers nouveaux modules ceux du package `express` (un module fournissant d'autres outils pour la création de serveurs), il faut d'abord choisir un répertoire qui sera la racine du répertoire contenant ces modules.

   La commande `npm install express` crée un nouveau répertoire `node_modules`, télécharge le package `express` et copie ses modules dans ce répertoire.

   Si l'on souhaite télécharger d'autres packages et centraliser leurs modules dans le même répertoire, il suffit de revenir dans le même répertoire que `node_modules` et de réexécuter la commande `nmp install` avec un autre nom de package.

   Si l'on souhaite se servir de ces modules avec Node dans un autre répertoire, il suffit de créer dans cet autre répertoire un lien symbolique vers le répertoire `node_modules` via la commande `ln -s chemin/vers/node_modules`.

2. **Globalement.** La commande `npm install -g express` copie les modules d'express non pas dans le répertoire où elle est lancée, mais dans un répertoire caché à la racine du répertoire de l'utilisateur (*e.g.* `/home/utilisateur/.npm-packages/lib/node_modules`). Tous les modules des packages téléchargés seront aussi installés dans ce répertoire avec la même option.

   Si l'on souhaite se servir de ces modules avec Node dans un autre répertoire, on peut bien sûr créer un lien symbolique, mais plus simplement ajouter à l'environnement une variable `NODE_PATH` en lui donnant la valeur renvoyée par la commande `node root -g`, renvoyant un chemin absolu vers le répertoire d'installation global de `npm`, par exemple en ajoutant au fichier `.profile` la ligne suivante :

   ```
   NODE_PATH=`npm root -g`
   ```

La première méthode est plutôt celle conseillée lorsque l'on travaille avec des versions différentes de packages, mais dans le cadre de ce cours, la seconde suffit. Les packages dont nous aurons besoin ici sont : `pg` (PostgreSQL), `express`, et `ejs` (embedded JavaScript).

# 2 Accès à une base de données

L'accès à une base de données PostgreSQL peut se faire avec les modules du package `pg`, dont le module principal porte le même nom.

Sur la page d'accueil de `pg`, l'exemple élémentaire illustrant l'usage du module n'est pas à suivre : il ne reflète qu'un usage naïf du module, la toute première erreur de cet exemple étant qu'un mot de passe ne doit *jamais* être écrit en dur dans du code, en particulier si ce code est collaboratif ou pire, sur un dépôt publique – au minimum, il est à stocker dans un fichier externe et privé, ou encore dans une variable d'environnement.

L'usage de `pg` présenté ici est plutôt celui que nous préconisons, aussi bien en termes d'efficacité que de lisibilité du code : usage d'un pool de connection, invocations de méthode en `await` dans des fonctions `async`.

## 2.1 Création d'un pool de connection

Un *pool de connection* permet de maintenir ouverte une unique connection à une base de données, qui peut être partagée par différentes parties du code. Cette connection centralisée évite d'une part de connecter plusieurs clients à un même serveur au risque de saturer le serveur, d'autre part de charger un unique client d'effectuer toutes les requêtes séquentiellement, au risque de perdre en efficacité. La création d'un pool peut se faire sur le modèle suivant :

```
const pg = require('pg');
const pool = new pg.Pool({
  user: 'moi',
  host: 'localhost',
  database: 'agenda',
  password: ...,       // accès à une information externe
  port: 5432
});
```

## 2.2 Accès à la base via le pool

La demande d'accès à la connection d'un pool est une opération asynchrone renvoyant une *promesse* qui, si elle est tenue, sera associée un objet de classe `Client` accédant à la base par l'intermédiaire du pool : elle se fait par invocation de la méthode `connect` du pool. La méthode `release` du client permettra de signaler au pool que le client ne souhaite plus se servir de sa connection.

Il faut évidemment attendre que cette promesse soit tenue pour disposer du client. Cela peut se faire par un `then`, mais en pratique, il est plus simple syntaxiquement d'invoquer cette méthode avec un `await` dans une fonction `async` :

```
async function operations() {
  const client = await pool.connect();
  // requêtes, traitement...
  // ...
  // dernière étape indispensable, libérer le client :
  client.release();
  // retour facultatif d'un résultat :
  return resultat;
});

operations()
.then(resultat => { ... })
.catch(err => console.err (err.stack););  // si une erreur se produit.
```

## 2.3 Requêtes des clients

La méthode `query` d'un client permet de spécifier une requête à la base sous la forme d'une simple chaîne de caractères. Elle renvoie dans ce cas une requête qui, si elle tenue, sera associée à un objet représentant le résultat de la requête.
Toujours pour des raisons de simplicité syntaxique, il vaut mieux invoquer cette méthode avec un `await` dans une fonction `async` :

```
async function operations() {
  const client = await pool.connect();
  // attente du résultat de la requête :
  let res = await client.query ("SELECT * FROM rendezvous");
  // chaque nom de colonne correspond à un nom de propriété de res :
  for (row in res) {
    console.log(res.date);
    console.log(res.heure);
    console.log(res.lieu);
  }

  // ...
  // libération du client :
  client.release();
  // retour facultatif d'un résultat :
  return resultat;
});

operations()
.then(resultat => { ... })
.catch(err => console.err (err.stack););  // si une erreur se produit.
```

# 3 Serveurs Express

Express est un package permettant d'assemble un serveur formé d'un mécanisme d'aiguillage de requêtes, dirigeant celles-ci vers un ou plusieurs gestionnaires en fonction des méthodes (`GET`, `POST`...) et des URI de ces reqûetes (leurs chemins, *e.g.* `/user/homepage`).

La terminologie de la documentation d'Express privilégie le terme d'*application Express* plutôt que de "serveur Express", et nous utiliserons ici la même convention. Une application Express se construit en trois étapes : la création de l'application, la définition de son mécanisme d'aiguillage – ce qu'on appelle des *routes* – puis son lancement :

## 3.1 Création d'une application Express

La création d'une application Express est très simple. Elle se fait sur le modèle suivant, dont nous préciserons le sens des éléments :

```
// création de l'application
const express = require('express');
const app = express();
const port = 8080;
// création d'une unique route pour les requêtes en GET vers / :
app.get('/', (req, res) => {
  console.log(res.body);
  res.send('Hello World!')
});
// lancement
app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
});
```

## 3.2 Les routes

**Définitions et principes.** Pour décrire le fonctionnement du mécanisme d'aiguillage des requêtes dans une application Express, nous utiliserons les définitions suivantes :

- Une *route* est l'association d'un ensemble de méthodes et d'URIs à un gestionnaire de requêtes. Cet ensemble est appelé le *domaine* de la route. Le gestionnaire sera toujours de la forme `(req, res, next) => {...}`[1].
- Les requêtes *acceptées* par une route sont toutes celles dont la méthode HTTP et l'URI font partie de son domaine.
- *Soumettre* une requête à une route consiste à déterminer si cette route accepte ou non la requête.

Avant de décrire la syntaxe de la création effective de ces associations dans une application, précisons un point important. L'*ordre* dans lequel sont créées ces routes détermine entièrement la manière dont l'application réagit à la réception d'une requête :

1. L'API est les exemples utilisent plutôt la notation `function(req, res, next)` , mais dans les deux cas, `this` est objet vide : le choix de l'une ou l'autre notation semble donc neutre.

1. Lorsqu'une requête HTTP est reçue par l'application, elle est soumise à chacune des routes dans l'ordre de leur création, jusqu'à ce que ce que soit trouvée la première route acceptant la requête. La réception d'une requête qui n'est accepté par aucune route déclenche la levée d'une exception.
2. Lorsqu'une route accepte une requête, son gestionnaire est appeleé avec `req` et `res` représentant la requête et la future réponse du serveur. Le gestionnaire peut compléter cette réponse de manière appropriée, puis demander :
   (a) ou bien qu'une réponse HTTP soit renvoyée au client,
   (b) ou bien que la requête soit soumise au routes suivantes, par un appel de `next()`.

Dans le second cas, le gestionnaire de la prochaine route acceptant la requête recevra la future réponse dans l'état où le gestionnaire précédent l'a laissée.

**Création de routes.** La création d'une route peut se faire à l'aide d'invocations de méthodes de formes suivantes :

```
// création d'une route acceptant les requêtes en GET et en uri :
server.get(uri, (req, res, next) => { ... });
// création d'une route acceptant les requêtes en POST et en uri :
server.post(uri, (req, res, next) => { ... });
// création d'une route acceptant toutes les requêtes en uri :
server.all(uri, (req, res, next) => { ... });
// création d'une route acceptant toutes les requêtes :
server.use((req, res, next) => { ... });
```

**Route finale.** Nous avons mentionné ci-dessus le problème posé par les requêtes qui ne sont acceptées par aucune route. Pour éviter une levée d'exception - et donc la terminaison de l'application - il est nécessaire après la création des différentes routes de finir cette construction par la création d'une route finale qui acceptera toutes les requêtes qui n'auront pas été acceptées par les précédentes :

```
// routes ...
// route finale : l'argument next est ici ignoré
server.use((req, res) => {
  // gestion des requêtes non attendues
});
```

**Spécification des URIs acceptées.** La forme de l'argument `uri` dans les invocations de méthodes ci-dessus peut spécifier l'acceptance d'une unique URI littérale (*e.g.* `'/blog/message'`), mais aussi d'un *ensemble* d'URIs à l'aide d'expressions régulières ou encore de paramètres de routes, des noms précédés de doubles points `':'` et désignant des sous-parties d'URIs. Ces sous-parties pourront être examinées dans le gestionnaire de la route via les propriétés de mêmes noms de `req.params` :

```
// requête vers : http://localhost:8080/blog/2021/decembre/commentaires
server.get('blog/:annee/:mois/commentaires', (req, res, next) => {
  // req.params.annee == '2021'
  // req.params.mois  == 'decembre'
});
```

## 3.3 Requêtes et réponses d'Express

Les requêtes et réponses dans Express sont respectivement des objets de prototypes `http.IncomingMessage.prototype` et `http.ServerResponse.prototype` (*c.f.* le chapitre précedent). Ces objets héritent donc de tous les événements, champs et méthodes des deux classes correspondantes, mais en pratique, l'usage des routes d'Express évite la prolifération de fonctions de rappel pour lire les données d'une requête ou construire sa réponse.

## 3.4 Extraction des données d'une requête

**Requêtes en `POST`.** Le type du contenu du corps d'une requête HTML en `POST` est spécifiée par l'entrée `Content-Type` de son header, par exemple :

1. `Content-Type: text/plain`
   exemple : `Hello World!`
2. `Content-Type: application/json`
   exemple : `{'message' : 'Hello World!', 'author' : 'me'}`
3. `Content-Type: application/x-www-form-urlencoded`
   exemple : `message=Hello+World!&author=me`

Le troisième type est par exemple émis par un formulaire HTML à deux champs textes de noms `message` et `author`, dont `Hello+World!` et `me` représentent les contenus textuels. Le contenu donné en exemple est parfois appelé *chaîne d'interrogation* ("query string").

Dans chaque cas, l'accès au contenu du corps d'une requête dans le gestionnaire d'une route ne peut se faire qu'en créant avant cette route un *parser*, une route en `use` dont le gestionnaire a pour seule fonction d'extraire ce contenu avant d'invoquer `next`. L'objet `express` possède trois méthodes permettant de générer des gestionnaires *ad hoc* pour chacun des types de contenus ci-dessus :

```
// 1. parser pour les requêtes en text/plain
app.use(express.text());
// 2. parser pour les requêtes en application/json
app.use(express.json());
// 3. parser pour les requêtes en application/x-www-form-urlencoded
app.use(express.urlencoded({extended : true}));
```

Le sens du `{extended : true}` dans le code ci-dessus est sans importance, il n'a de sens que pour Express. L'ajout de plusieurs parsers est neutre pour le fonctionnement de chaque parser : un parser ne décodera que le contenu de l'unique type pour lequel il est spécialisé, et transmettra sinon simplement la requête à la route suivante.

Les gestionnaires suivants pourront ensuite accéder au corps de la requête via la valeur de `req.body` : il s'agit pour le premier type de contenu d'une chaîne de caractères, pour les deux autres d'un objet littéral :

```
// exemples de valeurs de req.body pour les gestionnaires suivants :
// 1 : 'Hello World!'
// 2 : {message : 'Hello World', author : 'me'}
// 3 : {message : 'Hello World', author : 'me'}
```

En cas d'absence de parser approprié pour le contenu d'une requête, un gestionnaire recevra un objet `req` de propriété `body` indéfinie.

**Requêtes en `GET` + chaîne d'interrogation.** Lorsque l'URI d'une requête est complétée par une chaîne d'interrogation, *e.g.* `/forum?message=Hello+World!&author=me`, les paramètres de cette chaîne et leurs valeurs peuvent être récupérées dans un gestionnaire via `req.query` sous la forme d'un objet littéral, à condition que sa route soit précédée d'un parser pour le type de contenu `application/x-www-form-urlencoded`.

## 3.5 Construction de la réponse à une requête

Comme indiqué ci-dessus, les méthodes de `http.ServerResponse.prototype` peuvent être utilisées pour construire une réponse, éventuellement en plusieurs étapes et sous la responsabiité de routes distinctes par des appels de `next` : méthodes `write`, `end`, autres méthodes permettant de spécifier le header, etc.

Express propose cependant une autre méthode, `send`, qui exclut l'usage de `write` et `end` et qui n'est invocable qu'une et une seule fois sur l'argument `res` d'un gestionnaire. Cette méthode prend en argument une représentation du corps de la réponse qui sera renvoyée au client. L'avantage de son usage est qu'elle initialise automatiquement les headers de la réponse en fonction de cette représentation (`Content-Type`, `Content-Length`, ...) :

1. Lorsque l'argument de `send` est une chaîne de caractères, le corps de la réponse est cette chaîne de caractères :
   (a) Si cette chaîne est reconnue comme du contenu HTML (avec une certaine tolérance sur les noms de balises fermantes), le `Content-Type` du header prend la valeur `text/html`.
   (b) Sinon, le `Content-Type` du header prend la valeur `text/plain`.
2. Lorsque l'argument de `send` est un objet, le corps de la réponse est la conversion de cete objet en JSON par la méthode `JSON.stringify`, et le `Content-Type` du header prend la valeur `application/json`.

L'invocation de `send` peut bien sûr être encore précédé d'invocations configurant les autres parties de la réponse, par exemple son statut. En alternative à la méthode `send`, on peut mentionner :

- `res.sendFile`, renvoyant au client un fichier, le `Content-Type` étant alors déterminé par son extension.
- `res.redirect` permettant de rediriger le client vers une autre page.

# 4 Embedded Javascript

"Embedded Javascript" (EJS) est un package de Node utilisable conjointement avec Express, et permettant à un serveur de renvoyer une page HTML construite dynamiquement à partir d'un modèle de page (template) contenant du code Javascript chargé de compléter ses parties manquantes. Le nom du module EJS pour `npm install` est `ejs`.

Un modèle EJS s'écrit en HTML. Une couple de balises spéciales peut encadrer du code ou des portions de code Javascript, qui peut librement faire référence aux propriétés et méthodes d'un objet global non spécifié. Un second couple de balises spéciales permet de demander explicitement l'insertion dans la page d'une valeur d'expression Javascript, en particulier une valeur de propriété de cet objet global.

Le serveur, au moment où la page HTML est rendue, doit simplement indiquer quel est l'objet qui sera celui considéré comme objet global dans le code JavaScript du modèle EJS.

## 4.1 Un exemple simple

Supposons que l'on souhaite écrire un serveur permettant d'accéder au catalogue d'un label de disques. Le client doit pouvoir spécifier par une URI l'artiste dont il souhaite connaître la discographie, et la réponse du serveur doit une page web d'aspect uniforme quel que soit l'artiste choisi.

**Côté modèle.** Par convention les noms de fichiers de modèles EJS ont pour extension `ejs`, par exemple `artiste.ejs`. Ils doivent être placés dans un répertoire `views`, sous-répertoire de celui du code-source du serveur. Un exemple très simplifié de modèle EJS pour la spécification précédente pourrait être :

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Artiste :<%= artiste %></h1>
    <p>Albums :<p>
      <ul>
        <% let i;
           for (i = 0; i < albums.length; i++) { %>
              <li><%= albums[i] %></li>
           <% } %>
      </ul>
  </body>
</html>
```

Les couples de balises spéciales mentionnées ci-dessus sont `<%  %>` entre lesquelles se trouve du code Javascript, et `<%=  %>` permettant d'insérer une valeur d'expression dans l'HTML : `<%= artiste %>`, `<%= albums[i] %>`.

Les noms `artiste` et `albums` ne correspondent à aucun élément local du code JavaScript du modèle : il s'agira donc de propriétés de l'objet global choisi par le serveur au moment de la création de la page.

**Côté serveur.** Voici, à nouveau très simplifié, le code du serveur :

```
let catalogue = [];
catalogue[0] = {
  artiste : "Pink Floyd",
  albums : ["Dark side of the moon", "Wish you were here"]
};
catalogue[1] = {
  artiste : "Jimi Hendrix",
  albums : ["Electric Landyland", "Band of Gypsys"]
};

const express = require('express');
const server = express();
const port = 8080;
server.set('view engine', 'ejs');
server.get('/:num', (req, res) => {
  res.render('cours.ejs', catalogue[req.params.num]);
});
server.listen(port);
```

Les deux instructions-clefs sont ici :

1. `server.set('view engine', 'ejs');`,
2. `res.render('cours.ejs', catalogue[req.params.num]);`

La première spécifie un moteur de rendu pour la réponse (`ejs`). La seconde produit et renvoie un page web construite à partir du modèle de page `cours.ejs` et de l'objet `catalogue[req.params.num]`, qui est bien un objet à deux propriétés `artiste` et `albums`.

Noter la manière simpliste dont on accède au éléments du catalogue : `/0`, `/1`..., sans aucune vérification de la validité de l'index. On peut évidemment faire mieux.