ExpressJS

A - Découverte et Installation

1 - ExpressJS : présentation et grands principes de fonctionnement

Express JS est un framework (cadre de travail) Node.js de taille limitée et proposant de nombreuses fonctions pour simplifier la réalisation d'un site web. Express JS facilite l'organisation de l'application grâce à plusieurs notions :

```
- guide dans la gestion de la requête et la réponse
- pile de middlewares
- routages
- templates
```

2 - Installation d'ExpressJS

Après avoir créé un dossier spécifique, nous exécutons les deux commandes suivantes dans ce dossier :

a) Préalable : création d'un package JSON

```
$ npm init
```

Il s'agit de créer un fichier JSON (package.json) contenant une description de l'application Express JS créée (nom, version, fichier de départ …). L'ensemble des spécifications sont données dans la documentation npm.

b) Installation du module ExpressJS

```
$ npm install express
```

Cette commande va installer le module Express dans notre projet et ajouter sa mention dans le fichier package.json comme dépendance.
Ainsi, il est possible de déployer un projet réalisé, même si les modules ne sont pas présents, en exécutant la commande suivante au niveau du dossier racine du projet (le fichier package.json doit être présent) :

```
$ npm install
```

3 - Premier affichage

Après avoir écris ce code :

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

app.get('/', function (req, res) {
  res.send('Bonjour à tous !');
});

var server = app.listen(8080, function () {
  var addresseHote = server.address().address;
  var portEcoute = server.address().port;

  console.log('L\'application est disponible à l\'adresse
  http://%s:%s', addresseHote, portEcoute);
});
```

exécutons le :

```
$ node premiereEssai.js
```

Après avoir saisi localhost:8080 dans la barre d'adresse de notre navigateur, celui-ci affiche « Bonjour à tous ».

B - La gestion des routes et des fichiers statiques

1 - La gestion des différentes routes

À chaque fois qu'une requête est envoyée au serveur, elle traitée par celui-ci comme une route.
Voici deux routes différentes :

- http://localhost:8080/
- http://localhost:8080/informations

La première est gérée par ExpressJS de la façon suivante :

```
app.get('/', function (req, res) {
  res.send('Bonjour');
});
```

et la seconde ainsi :

```
app.get('/informations', function (req, res) {
  res.send('Une information supplémentaire');
});
```

Ces deux routes se placent entre l'appel du module ExpressJS (require …) et l'écoute du serveur (listen …).

2 - Envoyer des fichiers statiques au client

a) Cas général

L'application web permet de générer des fichiers html en fonction des requêtes qui lui sont envoyées; mais également de fournir des fichiers statiques stockés sur le serveur tels les images, les CSS ou le JavaScript par exemple.
Express permet de simplifier l'écriture des URL pour ces fichiers en les stockants tous dans un dossier parent.
Il faut utiliser, pour ce faire, une fonction middleware (voir plus bas pour cette notion) en écrivant :

```
  app.use(express.static('public'));
```

Vous pouvez désormais stocker tous vos fichiers dans le dossier public, situé à la racine de votre application, et écrire les URL comme suit :

```
http://localhost:8080/images/kitten.jpg
http://localhost:8080/css/style.css
http://localhost:8080/js/app.js
http://localhost:8080/images/bg.png
http://localhost:8080/hello.html
```

Vous pouvez créer de multiples dossiers statiques :

```
app.use(express.static('public'));
app.use(express.static('files'));
```

Express cherchera successivement vos fichiers dans les différents dossiers statiques indiqués. Ceci nous permet, par exemple, d'envoyer une balise image au client :

```
app.get('/image', function(req, res) {
  res.send('<img src="public/paysage.jpg">');
});
```

Sur le serveur, l'image est présente dans le dossier public qui a été, préalablement, déclaré au moyen d'un app.use.

b) Les fichiers HTML

Pour l'instant, nous envoyons une chaine de caractères en utilisant la méthode send de l'objet res fourni par Express comme second argument de la fonction de retour.
Il est possible, également, d'envoyer un document html enregistré sur le serveur. Ce fichier est stocké dans un dossier qui sera déclaré par un app.use (fichiers dans l'exemple ci-dessous)
Ensuite, nous pouvons écrire dans une route :

```
app.get('/page', function(req, res) {
  res.sendFile('index.html',{root:'fichiers'});
});
```

C - Un moteur de template : Pug

1 - Principe de base

Pug est un langage de templace. Cela permet de séparer le code qui génère l'affichage du reste du code JavaScript. Il faut obligatoirement utiliser la syntaxe du langage.
Nous insèrons le code suivant dans le fichier JavaScript

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

//définit le dossier de base pour les fichiers statiques (css, images, js,
app.use('/static', express.static('fichiers'));

//permet d'utiliser Pug
app.set('view engine', 'pug');

//redéfinir le dossier recevant les fichiers .pug (par défaut : views)
app.set('views','mestemplates');

//chemin de base de mon application :
app.get('/',function (req,res) {
  //permet le rendu du fichier index.pug qui donne un html envoyé ensuite a
  res.render('index', {title: 'Titre de la page',
    message : 'Du texte dans la page'});
});
```

Nous créons un fichier index.pug dans le dossier mestemplates.

```
doctype html
head
  title= title
body
  h1= message
```

2 - L'inclusion

Dans nos pages web, une partie de la structure est commune (en-tête, menu …). Il est donc préférable de séparer ces parties communes du reste. Prenons, tout d'abord, un fichier index.pug

```
doctype html
html
  include ./includes/head.pug
  body
    h1= titreH1
    p Un texte court
  include ./includes/foot.pug
```

Nous utilisons ensuite les fichiers head.pug et foot.pug situés dans le dossier includes lui-même situé dans le dossier mestemplates.
Le fichier head :

```
head
  title= titrePage
  script(src='script.js')
  link(rel="stylesheet", href="style.css")
```

Le fichier foot :

```
p
  a(href='mailto:moi@maboitemail.com') Me contacter
```

L'ensemble nous donne un fichier html comple qui est envoyé au client.

3 - L'héritage (utilisation d'un layout)

Une autre technique de séparation du code consiste à utiliser un Layout. En premier lieu, prenons le fichier layout.pug

```
doctype html
html
  head
  block title
    title Default title
  link(rel="stylesheet", href="style.css")
  script(src="script.js")
  body
    block content
```

Si nous utilisons le fichier index.pug ci-dessous, nous obtenons une page complète.

```
extends ./layout.pug

block title
  title= titrePage

block content
  h1= titreH1
  p Un texte court
```

Chaque fichier .pug appelé fera appel au layout.

D - Améliorations

1 - Gestion des valeurs avec la méthode GET

Nous pouvons passer des valeurs dans l'url et les récupérer côté serveur. Il existe deux façons de faire.

a) req.params

Nous utilisons l'adresse suivante pour la requête: http://localhost:8080/infos/truc/machin.
Nous pouvons mettre en place la route suivante pour récupérer les valeurs truc et machin :

```
app.get('/infos/:un/:deux', function(req,res){
  console.log(req.params.un); //affiche truc
  console.log(req.params.deux); //affiche machin
});
```

b) req.query

Nous utilisons l'adresse suivante pour la requête: http://localhost:8080/question?r=chose&t=bidule.
Nous pouvons mettre en place la route suivante pour récupérer les valeurs chose et bidule :

```
app.get('/question', function(req,res)){
  console.log(req.query.r); //affiche chose
  console.log(req.query.t); //affiche bidule
}
```

2 - Les middleware

Express est une librairie web légère facilitant la contruction de routes et utilisant des fonctions middleware. Une demande expresse est, avant tout, une série d'appels de middleware.
Le middleware est une fonction qui va être appelée entre la requête et la réponse pour apporter des possibilités supplémentaires à notre code.
Prenons le code suivant :

```
app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
```

À chaque requête, l'heure sera affichée dans la console. Vous remarquez que la fonction middleware a pour fondement l'écriture app.use(). On donne comme argument une fonction.
Parmi les middleware les plus habituelles avec Express, on trouve la définition de dossiers statiques, l'usage d'un dossier spécifique pour le template, le traitement des requêtes en POST ou encore la gestion des sessions.

3 - Gestion des valeurs avec la méthode POST

Les requêtes de type POST demandent une gestion particulière pour traiter les valeurs envoyées. Voici la partie centrale du template pug pour générer un formulaire en html:
Prenons le formulaire suivant :

```
form(action='reponse' method='post')
  input(type='text' name='texte')
  input(type='submit' value='Envoyer')
```

Pour traiter convenablement les données envoyées dans la requêtes, nous allons utiliser le module body-parser ce qui nous donne le code suivant :

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');
//Support des envois
app.use(bodyParser.urlencoded({
  extended: false
}));

app.set('view engine', 'pug');

app.post('/reponse', function (req, res) {
  res.render('reponse',{ titrePage: 'Réponse du formualaire',
  titreH1: 'Une page de réponse', reponse: req.body.text});
});
```

Nous remarquons que le champ texte du formulaire est récupéré à la fin de la ligne 12 (req.body.text).
Enfin, voici la partie centrale du template de l'affichage de la réponse :

```
h1= titreH1
p La réponse du formulaire est :
p #{reponse}
```

4 - Gestion des erreurs

Par défaut, Node.js ne génère par les erreurs. Il nous faut donc envoyer une page avec un en-tête 404 lorsque l'url demandée ne correspond à aucune page du site. Dans le code suivant, situé après toutes les autres routes, nous ajoutons l'en-tête 404 avant de faire le rendu de la page 404 créée par ailleurs.

```
app.use(function(req, res, next){
  res.status(404).render('404',{ titrePage: 'Erreur', titreH1: 'Des erreurs
});
```

L'erreur 404 n'est pas la seule présente. Il y, par exemple, l'erreur 503 si le serveur ne fonctionne pas correctement ainsi que l'erreur 403 pour un contenu protégé.
Prenons cet extrait d'une route :

```
app.get('/demande', function(req, res, next) {
  …
   if (err) {
     res.status('503');
     next();
     return;
   }
  …
});
```

Il ne faut pas oublier de prévoir l'argument next pour chaque route. Ainsi, nous arrivons dans le middleware ci-dessous situé après toutes les routes :

```
app.use(function(req, res, next){
  switch(res.statusCode){
    case 403:
      res.render('403',{ title:'Accès interdit !' });
    break;
    case 503:
      res.render('503',{ title:'Problème'});
    break;
    default:
      res.status(404).render('404',{ title:'Page inconnue' });
  }
});
```

Le test se fait sur le code d'erreur (statusCode) présent dans l'objet res pour afficher une page différente selon le type d'erreur.

5 - Les sessions

Une session est une période délimitée pendant laquelle un client est en communication avec un serveur qui réalise des opérations pour ce client identifié de façon individuelle. Cette identification permet de restreindre l'utilisation d'une partie du site à certaines personnes, enregistrer des données personnelles …
Nous devons utiliser ici deux modules : cookie-parser et express-session. Prenons le code suivant :

```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');

var app = express();
var sess;
app.use(cookieParser())
app.use(session({
    secret:'123456789SECRET',
    saveUninitialized : false,
    resave: false
}));

app.set('view engine', 'pug');
app.use('/static',express.static('files'));

app.get('/', function(req,res){
  sess = req.session;
  if (sess.nbr) {
    sess.nbr++;
  } else {
    sess.nbr = 1;
  }
  res.render('index',{titre : 'accueil',
  nombre : sess.nbr});
});

app.listen(8080,function(){
  console.log('écoute sur le port 8008');
});
```

Le template Pug suivant permet d'afficher à l'utilisateur le nombre de fois qu'il affiche la page :

```
doctype html
head
  title= titre
body
  h1 Accueil du site
  p Un texte
  p Vous avez affiché cette page #{nombre} fois.
```

E - Connection à la base de données

1 - Principe

Dans le code suivant, nous nous connectons à la base de donnée. Si cette connexion ne renvoit pas d'erreur, le serveur se met en écoute et l'objet lié à la base de données est disponible pour être utilisé dans chaque route.

```
var express = require('express');
var app = express();
var MongoClient = require('mongodb').MongoClient;
var mongoClient;
app.set('view engine', 'pug');
app.set('views','pugFiles');

const urlDb = 'mongodb://localhost:27017';
const nameDb = 'blog';
const urlConnect = '8080';


app.get('/', function(req, res, next) {
  var collection = mongoClient.collection('articles');
  collection.find().sort({date:-1}).limit(10).toArray(function(err,data) {
    res.render('index', { title: 'Bonjour', h1: 'Une liste', liste: data})
  });
});

MongoClient.connect(urlDb, function(err, client) {
  if (err) {
    return;
  }
  mongoClient = client.db(nameDb);
  app.listen(urlConnect, function() {
    console.log(' - Le serveur est disponible sur le port 8080');
  });
});
```

2 - Sécurisation de la connexion

Nous ne sommes pas certain que la connexion va être disponible constamment. Nous devons donc réaliser la connexion dans chaque route et fermer la connexion à la fin de la route.

```
var express = require('express');
var app = express();
var MongoClient = require('mongodb').MongoClient;
var maDb;
app.set('view engine', 'pug');
app.set('views','pugFiles');

const urlDb = 'mongodb://localhost:27017';
const nameDb = 'blog';
const urlConnect = '8080';

app.get('/', function(req, res, next) {
  MongoClient.connect(urlDb, function(err, client) {
    if (err) {
      res.status(503);
      next();
      return;
    }
    var instance = client.db(nameDb);
    var collection = instance.collection('articles');
    collection.find().sort({date:-1}).limit(10).toArray(function(err,data) {
      mongoClient.close();
      res.render('index', { title: 'Bonjour', h1: 'Une liste', liste: data})
    });
  });
});

  app.listen(urlConnect, function() {
    console.log(' - Le serveur est disponible sur le port 8080');
  });
});
```

3 - Factorisation de la connexion

Il serait plus judicieux de regrouper le code de la connexion dans un module (db.js). Par exemple :

```
var urlDb = 'mongodb://localhost:27017';
var nameDb = 'blog';

exports.connectDB = function(req, res, next,cb) {
  if (this.mongoClient && this.mongoClient.isConnected()) {
    var instance = client.db(this.mongoClient);
    cb(instance);
  } else {
    MongoClient.connect(urlDb, function(err, client) {
      this.mongoClient = client;
      if (err) {
        res.status(503);
        next();
        return;
      }
      var instance = client.db(nameDb);
      cb(instance)
    });
  }
});
```

Notre fichier principal sera alors le suivant :

```
var express = require('express');
var app = express();
app.set('view engine', 'pug');
app.set('views','pugFiles');

var dbInterface = require('./db');
const urlConnect = '8080';

app.get('/', function(req, res, next) {
  dbInterface.connectDB(req,res,next,function(db) {
    var collection = db.collection('articles');
    collection.find().sort({date:-1}).limit(10).toArray(function(err,data) {
      res.render('index', { title: 'Bonjour', h1: 'Une liste', liste: data})
    });
  })
});


app.listen(urlConnect, function() {
  console.log(' - Le serveur est disponible sur le port 8080');
});
```