

Pluralsight : Node.js

Building Web applications with Node.js and Express

↳ Beaucoup d'infos déjà vues dans les cours précédents.

Getting started

↳ Déjà vu dans les cours précédents.

The first page

Setting up Express:

O

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

app.get('/', (req, res) => {
    res.send("Hello from my app.");
});

app.listen(3000, () => {
    console.log("listening on port 3000");
});
```

O

⇒ On a déjà vu le fonctionnement.

Running Express:

↳ Lancer le serveur avec ... \$ node nnn

↳ nom du fichier.

On peut accéder à l'app en utilisant le port de localhost
qu'on a défini dans app.listen().

Debugging options:

chalk:

↳ Package pour ajouter des couleurs dans les messages.

Pratique pour mieux visualiser certains messages qui on console.log.

ex: const chalk = require('chalk');

console.log ("Listening on port" + chalk.green("3001"));

Templating strings:

↳ Déjà vu!

ex: console.log ("Listening on port \${chalk.green('3001')}");

① \$ { } ②

 => String

 => Contenu dynamique qu'on insère avec \${ }.

② => Délimite le templating string.

• String peut être écrit sur plusieurs lignes.

Debug:

↳ Package qui nous permet d'écrire des messages de debugging au lieu d'utiliser des console.log.

const debug = require('debug') ();

 → nom qu'on donne pour le fichier / la portion de code. C'est l'endroit qui sera affiché devant les messages => Qui envoie le message

debug (),

↳ Ecrit le message (= string).

Commande
différente pour
Windows.

Pour avoir les messages de debug:

... \$ DEBUG = ~~??~~ mode ~~??~~

↓
↳ nom du fichier où on lance
De qui on veut voir les messages.
↳ (*) => tous !

ex: const debug = require('debug')('app');
debug("Hello World!");

↓
... \$ DEBUG = * mode app.js

↓

app Hello World!

↳ nom qui on a donné pour require

↓
... \$ DEBUG = app mode app.js

↳ On verra seulement les messages venant de app.

Morgan:

↳ Va permettre de recevoir des infos concernant notre traffic web.

const morgan = require('morgan');
app.use(morgan(~~???~~));

~~???~~ => détermine si on recevra beaucoup d'info ou pas.
(ex: 'combined', 'tiny', ...)

Quand on a une requête sur le serveur Morgan
fera un console.log de différentes info sur la requête.

Serving index:

On enverra une page HTML fixe avec express.static.

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

⇒ Express cherchera au path ~~''~~ un fichier valide à envoyer. Pour la root le fichier valide est index.html. Si aucun fichier valide n'est trouvé on effectue le app.get de l'url.

On peut utiliser le package 'path' (built-in) pour cibler un path.

Imagineons que nous mettons tous nos fichiers fixes dans un dossier 'public'.

ex:

```
const path = require('path');
app.use(express.static(path.join(__dirname, '/public')));
```

 { app.get('/', (req, res) => {
 res.send("Hello from my app.");
 }); }

* On crée un path en joignant __dirname (le path du fichier de code) et /public.

express.static envoie un/des fichiers valide se trouvant au path *. Si aucun fichier n'est valide on effectue le app.get *.

Setting up tooling

→ NPM scripts, Nodemon, variables environnementales.

⇒ Déjà vu

On peut dans package.json configurer nodemon.

```
"nodemonConfig": {  
  "restartable": "ns", → En tapant "ns" on  
  "delay": 2500          relance le serveur  
}                         → Attends 2500 ms avant  
                           de relancer.
```



Le restart du serveur après une modification d'un fichier fonctionne tjs !

Dans la configuration de nodemon on peut définir des variables d'environnement.

```
"nodemonConfig": {  
  :  
  "env": {  
    "PORT": 4000  
  }  
}
```



On y accèdera dans le code avec process.env. ↓
variable

ex: const PORT = process.env.PORT

⚠ On doit relancer nodemon quand on modifie la configuration. Restart le serveur ne suffit pas.

Templating engines → Permet l'utilisation de HTML dynamique.

Using EJS:

On met en général tout notre code dans un dossier 'src'.

On utilisera un sous-dossier 'vues' pour mettre toutes mes vues / mes templates.

app.set ('vues', 'src/vues') → nom variable
→ valeur

↳ Permet de définir

Pour EJS on va définir une variable qui sera le path de où sont les vues:

app.set ('vues', './src/vues');

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

↳ On indique quel Templating engine on utilise.

Tci : EJS.

Fichiers .ejs:

↳ Fichiers avec du HTML dynamique. Ils seront utilisés par EJS.

On définit du code dans ces fichiers avec <% %>

Quand on insère la valeur d'une variable on utilise

un '='. ⇒ ex: <% = title %>

↳ Insère la valeur de la variable 'title'.

Au lieu d'utiliser res.send() on utilise res.render():

res.render ('nom', {variables});

→ nom du fichier .ejs (↑ sans ".ejs")

→ objet contenant les variables qu'on utilise.

```
ex: app.get('/', (req, res) => {
    res.render('index', { title: 'Globomantics' });
});
```

↳ Render le fichier index.js et on utilise 'Globomantics' comme variable title.

⚠ Si une page statique est envoyée avant un render, ce dernier ne se fera pas.

↳ Comme pour le res.send() vu avant.



Passing data:

On peut envoyer différents types de variables avec res.render(), dont des array ou objets.

Imaginons qu'on envoie aussi un array et qu'on souhaite afficher les valeurs dans une liste.

```
app.get('/', (req, res) => {
    res.render('index', { title: 'Globomantics',
        data: ['a', 'b', 'c'] });
});
```

On va dans le fichier .ejs créer une boucle JS entre <% .. %> et écrire à chaque fois ... avec la valeur de l'array.

Problème: ... n'est pas du JavaScript.

Solutio: On ferme la <% .. %> et on en réécrit un dès qu'on réécrit en JS.

⚠ .map() très utile ici!

```
<html>
  :
<ul>
  <% data.map( (i) => {
    %> <li> <%= i %> </li> <%
  });
  %>
</ul>
  :
</html>
```

⇒ avec .map(), pour chaque valeur de data on l'écrit entre des balises ` ... ` en html.

Working with templates:

On mettra dans le dossier 'public' tous les fichiers html, CSS, images, ... statiques.

On mettra dans le sous dossier 'views' du dossier 'src' les fichiers ejs, c'ds le html dynamique.

Using routing to build multiple pages

Implementing navigation:

↳ Mises en place de liens dans le HTML pour accéder aux autres url.

On ajoute autant de app.get() qu'on veut pour accéder différentes url.

Imaginons qu'on ait plein d'urls ayant un chemin en commun on utilisera un Router.



Implementing a router:

Router => permet de gérer l'entière du code pour une route et ses "sous-routes".

```
const sessionRouter = express.Router(); ①
```

:

```
app.use('/sessions', sessionRouter); ②
```

① On définit sessionRouter comme étant un Router



② On indique que la route /sessions et toutes ses sous-routes utilisent le routeur sessionRouter.

On va ensuite définir toutes les routes à partir du routeur.

```
sessionRouter.route('/')
```

```
  .get ((req, res) => {
```

```
    res.send('Hello sessions');
```

```
});
```

```
sessionRouter.route('/id1')
```

```
  .get ((req, res) => {
```

```
    res.send('Hello session with id 1');
```

```
});
```

- ⚠ Quand on définit des routes à partir d'un routeur
on repart de la route sur laquelle on utilise ce Routeur.
↳ sessionRouter.route('/'). . .
↳ on définit la route /sessions
↳ sessionRouter.route('/id1'). . .
↳ on définit la route /sessions/id1

Rendering the page:

On peut à chaque route faire un render d'un code html.

⇒ Exemple sur la route /sessions

On envoie un fichier.ejs avec le html de la structure de base du site (header, footer, . . .) et on fait boucler un code html affichant à chaque fois une session.

On envoie parmi les variables un array d'objets.

Chaque objet représente une session (avec titre et description). Les infos des sessions sont insérées avec des <% . . . %>.

Pouring data:

On peut envoyer des données d'un json (techniquement la structure d'un objet).

Il faudra juste faire un require du fichier json et l'envoyer avec le render.

ex: :

```
const sessionsData = require('./src/data/sessions.json');  
//  
sessionRouter.route('/'). . .  
res.render('sessions', { sessionsData, });
```

On peut également passer des valeurs de variables dans des liens.

↳ ex: <a href="/sessions/<% = index %>"/>...

Creating a single item route:

⇒ Générer des URL dynamique (ex: /sessions/{id})

① sessionRouter.route('/':id)

.get((req, res) => {

const id = req.params.id;

res.send('Hello from session ' + id);

① En utilisant la notation ":/" on fait passer la valeur de / dans la fonction. Peu importe ce que c'est.

② On accède aux paramètres de la route avec req.params et on peut cibler un paramètre précis.

Rendering a single item:

↳ idem rendering d'une page.

Separate Router files:

On stocke généralement les fichiers de routes dans un sous-dossier 'routes' dans le dossier 'src'.

On isolera tout ce qui concerne un routeur dans son propre fichier.

Comment faire le lien? ⇒ Chercher!

node_modules
public
src

→ data
→ routers
↳ views

app.js
package.json

:

↓
structure
pour le moment

Connecting to a database

↳ On verra l'exemple avec MongoDB. Les autres DB sont normalement assez similaires.

Setting up MongoDB:

↳ mongodb.com

Il y a moyen d'avoir une DB MongoDB gratuitement sur le cloud (AWS, Azure, ...).

Creating admin routes:

| --- \$ npm install mongodb

const mongodb = require('mongodb');

On se connecte à la DB en utilisant :

mongodb.MongoClient.connect (URL);

→ URL de la DB. On doit y mettre le mot de passe

ex:

: : : : } tous les require, création de Routed, ...

adminRouter.route('/').get((req, res) => {

const url = ' ';

①

const dbName = 'globomantics';

(async function mongo() {

②

let client;

try {

client = await MongoClient.connect(url);

③

const db = client.db(dbName);

④

const response = await db.collection('sessions')

.insertMany(sessions);

⑤

```

    res.json(response); (4)
}

catch(error) {
    debug(error.stack);
}

})();
}
);

```

- ① On définit l'url de la DB et le nom de la DB.
- ② On définit une fn asynchrone et on l'appelle.

↳ (async function mongo() { ... }) ()

on définit on appelle

- ③ On se connecte à l'url.
- ④ On définit la db qui on va utiliser
- ⑤ On insère dans la collection 'sessions' de la db un json sessions (dont on a fait un require avant).
- ⑥ On envoie sous la forme d'un json les données qui on vient d'insérer.

()

⇒ Ressources !

Inserting sessions:

↳ Vérification de si l'exemple a fonctionné.

Selecting sessions:

↳ Récolter depuis la DB.

const response = db.collection('sessions').find().toArray();

↳ Récupère toutes les données de la collection 'sessions'.

On pourra ensuite envoyer ou faire un render.

Selecting a session:

↳ creation de routes qui vont chercher un document particulier dans la DB.

On crée une route normalement et on va utiliser :

```
db.collection('sessions').findOne({  
  _id: new mongodb.ObjectID(id) });
```

⇒ On cherche dans 'sessions' le document avec un _id égal à id (passé en paramètres).

⚠️ id est un string et _id un objet. On utilise
`new mongodb.ObjectID()` pour créer un
objet à partir de id et faire correspondre les types.

Securing your application