

Rappels et Devoir à rendre

Variables d'environnement

Les variables d'environnement servent à passer des données sensibles (mots de passe, apikeys, ...) ou pouvant changer facilement lors du développement d'un poste à un autre (adresse de connexion serveur, ...). **En aucun cas vous ne devez avoir ces informations en dure dans votre code.** Que ce soit dans le `package.json` ou dans un fichier de votre programme.

Ex : Tout de suite, vous bossez sur un repository git public. Pour la partie envoi de mail, vous aurez certainement des variables d'environnement pour faire la connexion SMTP. Voulez-vous que des gens aient accès à votre compte mail ?

Déclaration

- Les variables d'environnement utilisent la norme `CAPITALS` (de la forme `UNE_VAR_EXEMPLE`)
- En local, vous avez deux façons de les utiliser :
 - i. avant votre commande dans le shell : `MONGO_HOST=localhost:27017;MONGO_NAME=tpnode node server.js`
 - ii. via votre IDE. Webstorm et VSCode vous permettent de configurer des variables d'environnement pour le lancement de votre projet (mode recommandé)
- En production :
 - Si vous utilisez Docker, vous renseignez les variables disponibles dans le `DockerFile`. Ensuite vous les initialisez via par exemple le lanceur (`docker run`, `docker-compose.yml`, votre orchestrateur (Rancher, ...))
 - Si vous êtes directement sur un serveur, via les variables d'environnement système (`.bash_profile`, `.bashrc`, ...). Ou alors directement via l'appel de la commande :
`MONGO_HOST=localhost:27017;MONGO_NAME=tpnode node server.js`

Utilisation

Vous pouvez récupérer **toutes** les variables du serveur lançant le projet via le package `process` disponible de base dans Node.JS. De là vous pouvez les récupérer via `process.env`. Par exemple, si vous voulez récupérer la valeur de votre variable `MONGO_HOST`, vous avez juste à faire un `process.env.MONGO_HOST`.

Promesses

Les promesses permettent d'avoir une logique asynchrone beaucoup plus simple que ce qu'il se fait de base au sein de Node.JS.

Ex : De base, vous pouvez vous retrouver facilement avec un code comme ceci:

```
asyncFunc1(params, function (err, result1) {
  asyncFunc2(params, function (err, result2) {
    asyncFunc3(params, function (err, result3) {
      asyncFunc4(params, function (err, result4) {
        // ...
      })
    })
  })
})
```

Ce qui peut vite rendre votre programme illisible ou difficile à suivre. Le cas affiché plus haut s'appelle le [Callback Hell](#) (je vous invite à cliquer sur le nom pour avoir un article détaillé sur ça).

L'une des solutions (conseillé par beaucoup de monde) est de passer par les promesses au lieu de fonctions à callback. Cela implique beaucoup de changements au niveau du code de base et a beaucoup d'impacts en ce qui concerne la gestion des erreurs.

Avant de rentrer dans les détails, qu'est-ce qu'une fonction à callback ? Il faut savoir que Node.JS fonctionne en asynchrone et que, selon les fonctions appelées, on doit attendre le retour de cette fonction pour continuer. Les fonctions de callback servent à ça et une convention de déclaration de fonction a été faite :

```
var maFuncAsync = function (params, callback) {
  // traitement de la fonction

  // Si erreur
  callback(err); // préférez utiliser une instance d'Error

  // si tout va bien et que vous devez passer un résultat
  callback(null, result);

  // si tout va bien et que vous n'avez pas à passer de résultat
  callback();
}
```

Ce qui donne, en appel de cette fonction la chose suivante :

```
maFuncAsync({
  field : 'value',
}, function (err, result) {
  // ...
});
```

Mais les fonctions asynchrones posent différents problèmes en ce qui concerne la gestion des erreurs quand celles-ci sont déclenchées au sein de celles-ci. En effet, Node.JS ne sait pas faire remonter l'erreur au sein de la stack (essayez de faire un try/catch autour d'une fonction asynchrone, ça ne marchera pas) ce qui fait que votre programme va planter car c'est le *mainProcess* qui va la récupérer.

Pour les promesses c'est différent. Si on reprend notre exemple de callback Hell, voici ce que ça donnerai en promesse :

```
asyncFunc1(params)
  .then((result) => asyncFunc2(params))
  .then((result) => asyncFunc3(params))
  .then((result) => asyncFunc4(params))
  // ...
  .catch((err) => {
    // gestion des exceptions
  });
```

Beaucoup plus simple n'est-ce pas ? Mais comment sont faites les promesses ? Tout simplement comme ceci :

```
const maPromise = new Promise((resolve, reject) => {
  // si erreurs
  return reject(error); // préférez utiliser une instance d'Error

  // si tout va bien et qu'on veut passer un résultat
  return resolve(result);

  // si tout va bien et qu'on ne veut pas passer de résultat
  return resolve();
})
```

La fonction `then` est appelée automatiquement si le `resolve` est appelé dans votre promesse.

La fonction `catch` est appelée automatiquement si le `reject` est appelé dans votre promesse ou si une exception a été déclenchée dans toute la chaîne de promesses que vous avez.

Attention à ne surtout pas oublier les `return` pour que tout fonctionne bien.

Mais `Bluebird` dans tout ça ? Bluebird est juste une autre implémentation des promesses (utilisant la norme de celles disponibles nativement et du coup étant compatibles avec) rajoutant diverses fonctions utilisées aux promesses (`each`, `map`, ...) facilitant le codage de votre projet.

Hapi : Plugins et Handlers

Au sein du projet fil rouge, vous avez normalement commencé à travailler avec un plugin propre au projet pour toute la partie utilisateurs. Si je vous fais utiliser un plugin c'est pour plusieurs raisons :

1. Voir par vous même comment un plugin se créer et fonctionne
2. Modulariser votre code. Ce qui facilite les mises à jour, debug, ... En effet, si vous devez changer de library interne au plugin, de fonctionnement dans une méthode, ... vous avez juste à toucher à ce plugin (parfois au handler).

Pour garder cette séparation entre plugins il faut par contre suivre les règles suivantes :

- Un plugin ne doit pas appeler un autre plugin (hormis ceux propres à la partie log/bdd si besoin)
- Chaque fonction de votre plugin doit être soit synchrone, soit être une promesse (cette dernière afin de gérer au mieux les erreurs et éviter les Callback Hell)
- Pour communiquer entre vos plugins, vous devez passer par un handler.
- La fonction `register` définie au sein d plugin est appelée par Hapi lors du `server.register` et sert à initialiser votre plugin (configuration, méthodes disponibles, création de routes propres, ...)

Vous trouverez la structure de base des plugins à la fin du TP4

Ce qui devrait vous donner, par exemple, la chose suivante dans un handler :

```
const routeExample = (request, reply) {
  request.server.plugins.pluginA
    .findItem({
      id : request.params.id,
    })
    .then(item => request.server.plugins.pluginB.sendItem(item))
    .then(result => reply(null, result))
    .catch(err => reply(err));
}
```

TP Fil rouge : Choses à faire

- Récupération de tous les utilisateurs
- Récupération d'un utilisateur donné
- Ajout d'un utilisateur
- Modification d'un utilisateur
- Suppression d'un utilisateur
- Génération aléatoire de 100 utilisateurs
- Authentification d'un utilisateur
- Régénération du mot de passe utilisateur
- Utilisation du module créé au TP2 pour la génération du mot de passe
- Email de création de l'utilisateur
- Email de régénération du mot de passe
- Email de modification de l'utilisateur

La partie socket.io n'est plus à faire