Introduction à Node.js

I. Introduction

A. De JavaScript à NodeJS

Vous connaissez déjà JavaScript **coté client** avec notamment des frameworks comme ReactJS ou via ES5, 6, 7 et 8. Dans les cas évoqués, le JavaScript s'exécute dans le navigateur Web. Il permet de créer des pages web dynamiques, d'interagir avec les utilisateurs et de manipuler le contenu de la page.

A l'inverse Node.js est un environnement d'exécution de JavaScript **côté serveur**. Le code ne s'exécute donc pas dans le navigateur mais bien directement sur un serveur. Il a été créé en 2009 par Ryan Dahl et est bâti sur le moteur JavaScript V8 de Google (utilisé dans le navigateur Google Chrome). Il est à présent maintenu par la Fondation OpenJS.

Rassurer-vous ! Si Node.js apporte des fonctionnalités différentes, il utilise le même langage JavaScript que vous connaissez déjà.

B. Pourquoi Node.js

Node.js est un **système « single thread » non bloquant**. Ce qui offre la possibilité d'exécuter plusieurs requêtes vers le serveur en simultanées. Concrètement, cela signifie que Node.js est capable de gérer plusieurs requêtes sans attendre que la requête précédente soit terminée. Ceci est possible grâce à l'**asynchrone** géré par la librairie **libuv**. A l'inverse un langage comme PHP qui est synchrone traite les requêtes au fur et à mesure que les threads se libèrent.

> Pour aller plus loin : Vue d'ensemble du blocage et du non-blocage
>
> https://nodejs.dev/fr/learn/overview-of-blocking-vs-non-blocking/

Ce système permet de **gagner en performance** de manière incroyable. Le fait qu'il soit basé sur le léger et puissant moteur V8 de google participe aussi à cela.

Comme pour PHP, Node.js offre un bel environnement avec de nombreuses **librairies** ou encore **frameworks**. Il a également une très grande **communauté**, on trouve aisément du support.

Les applications Node.js sont plutôt **décomposées**, ce qui permet d'ajouter, modifier ou supprimer des modules facilement sans impacter d'autres fonctionnalités.

Un autre avantage est le langage utilisé. **Un développeur qui connait JavaScript peut alors développer back et front en toute autonomie**.

Notez aussi qu'il est bien plus facile de mettre en place des **applications en temps réel** comme les chats, les outils de collaboration ou encore les jeux en ligne avec Node.js.

C. L'event Loop

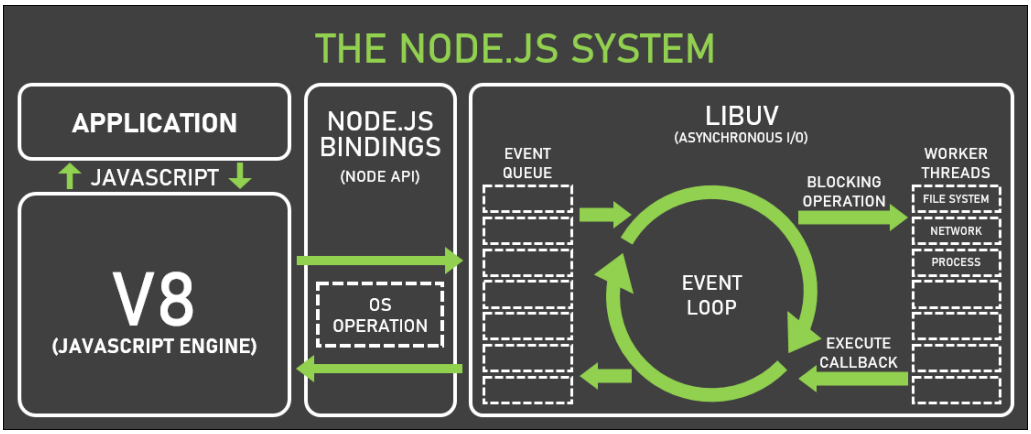


Figure 1: Schématisation de l'event loop de Node.js (source : https://www.tutorialandexample.com/node-js-event-loop)

Lien utile : https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick

II. Fonctions synchrones et asynchrones

Node.js est plus performant que d'autres langages car il permet les traitements asynchrones. Mais quelles différences avec les fonctions synchrones que vous avez utilisées jusqu'à présent ?

A. Fonction synchrone

Une fonction synchrone **bloque l'exécution** du programme jusqu'à ce son traitement soit terminée. Donc tant que cette fonction est en cours, les suivantes ne sont pas exécutées.

On les utilise pour des traitements immédiats et bloquants.

B. Fonction asynchrone

A l'inverse, la fonction asynchrone **ne bloque pas l'exécution** du programme. D'autres tâches peuvent alors continuer à s'exécuter en arrière-plan pendant que cette fonction est en cours d'exécution.

On les utilise généralement pour les traitements qui demande plus de temps comme les requêtes réseaux, les téléchargements de fichiers, etc.

1. Avec une fonction de rappel (callback)

Avec cette première possibilité, on appelle notre fonction en lui passant en paramètre une fonction de callback. Cette fonction de callback sera appelée dès lors que la fonction aura terminé son exécution.

Exemple :

```
//setTimeout est une fonction asynchrone qui affichera « Fonction
terminée » après 100ms.
setTimeout(() => console.log('Fonction 1 terminée'), 100);
setTimeout(() => console.log('Fonction 2 terminée'), 50);
```

On aura en sortie :

```
Fonction 2 terminée
Fonction 1 terminée
```

En effet, la fonction 2 étant plus rapide que la 1, même si elle est placée après, son exécution se terminera plus tôt. Les deux fonctions s'exécutent en parallèle.

Utiliser cette méthodologie peut causer certains problèmes :

Impossible de prédire quand une fonction aura terminé son exécution et donc dans quelle ordre les fonctions vont s'exécuter. Ceci ne pose bien sûr pas de problème si nous n'avons qu'une seule fonction asynchrones ou si les fonctions sont complètement indépendantes les unes des autres. Dans le cas contraire, si les fonctions dépendent les unes des autres, il va falloir imbriquer les callbacks ce qui rend très vite le code lourd et illisible. Sachant qu'il faut en plus ajouter à cela la gestion d'erreurs.

2. Avec une promesse (then/catch)

La deuxième possibilité, plus moderne et répondant aux problématiques des fonctions de rappel est le système de promesse (*promise*). Cette promesse peut être résolue (opération réussie – « *resolve* ») ou rejetée (opération échouée – « *reject* »). « .then » permet alors de traiter le résultat si la promesse est résolue tandis que « .catch » permet de traiter le résultat si elle est rejetée.

```
findWoods()
.then((woods) => {
      console.log('We have found some woods' : woods)
})
.catch((err)=>{
      console.log('An error occured') ;
});
```

Il est également possible de chainer les « .then » :

```
findWoods()
.then((datas) => {
      //…make some actions on datas…
      return datas ;
})
.then((woods) => {
      console.log('We have found some woods' : woods)
})
.catch((err)=>{
      console.log('An error occured') ;
});
```

3. Avec async/await

Ces deux mots clés sont plus récents et permettent de rendre encore plus lisible l'appel de fonctions asynchrones avec les promesses. Le code ressemble visuellement alors plus à du code synchrone. Afin d'intercepter les erreurs nous utiliserons un bloc try/catch.

Si on reprend l'exemple précédent, nous aurions :

```
try{
	const woods = await findWoods() ;
	console.log('We have found some woods' : woods)
} catch(err){
	console.log('An error occured') ;
}
```

« await » ne peut être utilisé qu'avec des fonctions async. Donc ici findWoods doit être « async ».

Si on utilise « await » au sein d'une fonction, celle-ci doit également être « async » :

```
async function renderWoods{
	try{
	    const woods = await findWoods() ;
	    console.log('We have found some woods' : woods)
	 } catch(err){
	    console.log('An error occured') ;
	 }
}
```

III. Installer et initialiser Node.js

A. Installation de Node.js et npm

ACTIVITE 1 : Installation de Node.js et npm

https://nodejs.org/fr/download

Vous pouvez vérifier la bonne installation avec `$ node -v` et `$ npm -v`

On peut aussi remplacer -v par --version.

B. Initialiser un projet

Pour initialiser un projet Node.js, il suffit de saisir la commande `$ npm init` dans le dossier du projet. Il vous faudra répondre à quelques questions. Lorsque la commande est terminée, vous verrez qu'un fichier package.json a été créé.

C. Le gestionnaire de paquet npm

Si PHP utilise composer, Node.js utilise *npm* (Node Package Manager).

1. Principales commandes

- `$ npm init` : pour initialiser un nouveau projet avec node.
  La commande vous guidera sur l'initialisation en vous posant différentes questions (nom du projet, description, auteurs, version, etc).
- `$ npm install nom_du_module` : pour ajouter une nouvelle dépendance au projet. Cette commande l'installe et l'ajoute au fichier package.json.
  On peut différencier l'installation pour la production ou le développement via les flags :
  - « --save-prod » pour la production.
  - « --save-dev » pour le développement.
- `$ npm uninstall nom_du_module` : pour supprimer une dépendance du projet. Cette commande la désinstalle et la retire du fichier package.json.
- `$ npm install` : pour télécharger et installer toutes les dépendances du projet détaillées dans le fichier package.json.
- `$ npm update` : pour mettre à jour toutes les dépendances du projet vers la dernière version compatible.
- `$ npm outdated`: pour vérifier s'il existe des dépendances obsolètes.

Pour aller plus loin : la liste complète des commandes

https://docs.npmjs.com/cli/v9/commands

2. Le site npmjs

Le site https://www.npmjs.com/ recense toutes les dépendances utilisables avec *npm*.

Prenons l'exemple du module *helmet* qui est d'une grande aide pour sécuriser une application Express, on retrouve :

- L'auteur
- Le lien GitHub
- Des chiffres sur les actions de la communauté : nombre d'installation, nombre de fork, nombre d'issues, etc.
- L'historique des versions
- Les prérequis
- Un README avec diverses informations comme les instructions d'installation ou encore une documentation ou différents liens.

C'est un outil plutôt pratique lorsqu'on a besoin de choisir une version ou un outil à utiliser au sein d'un projet.

3. Autoload

npm ne propose pas d'autoloader de façon native. Certains packages le permettent.

4. Les contraintes de version

npm permet de gérer les versions des bibliothèques externes, ce qui facilite la maintenance et la mise à jour du projet. En s'assurant que les bibliothèques externes utilisées par le projet sont à jour et ne contiennent pas de vulnérabilités, *npm* améliore la sécurité du projet.

Une version se compose de quatre éléments :

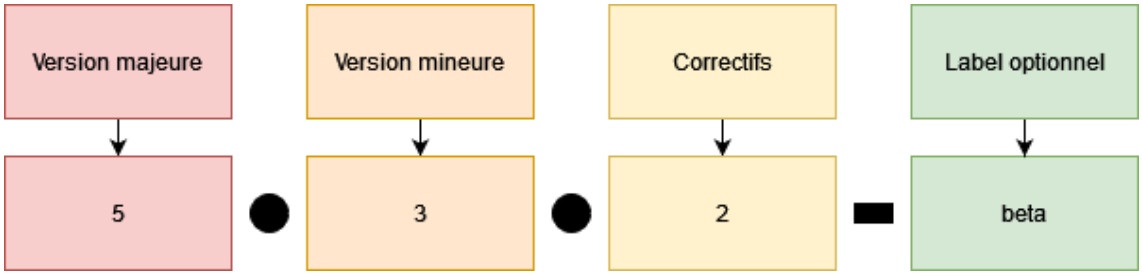


Figure 2: Découpage d'une version

On peut faire le choix d'une version fixe ou non. Voici comment choisir votre version :

- Pour la **version exacte** on donne simplement le numéro.
  Ex : « 5.3.2 »

- On peut utiliser les **opérateurs de comparaisons** >, >=, < et <=.
  Ex : « >4.4.2 »

- On peut utiliser une **plage de version** générique avec *.
  Ex : « 1.2.* » permet de sélectionner une version entre 1.2.0 et 1.2.

- On peut utiliser des **contraintes de version avancées** :
  - **~** permet les mises à jours mineures automatiques.
    Ex : « ~ 1.2.3» permet les mises à jour entre ~1.2.3 et 1.9.9
  - **^** permet les mises à jour mineures et majeures automatiques uniquement si la mise à jour ne casse pas la compatibilité.
    Ex : « ^ 1.2.3» permet toutes les mises à jour entre 1.2.3 et 1.9.9. On pourra également faire la mise à jour vers 2.0.0 si la compatibilité le permet.

5. Utilisation de scripts

npm permet en outre d'exécuter des scripts à différents moments du cycle de vie du projet. On les utilise pour automatiser certaines tâches comme le lancement d'un serveur, la compilation ou encore générer une documentation. On peut également créer des raccourcis pour exécuter des commandes en ligne de commande.

On ajoutera alors une section « scripts » dans le fichier package.json. On peut associer un script à différents évènements : https://docs.npmjs.com/cli/v9/using-npm/scripts.

Pour lancer un script non lié à un évènement manuellement, on utilisera la commande :

```
$ npm run nom_commande //ou npm nom_commande
```

ACTIVITE 2 : Prise en main de npm

- Initialiser un projet npm. Lors du choix du point d'entrée saisir « server.js »

- Créer le fichier server.js

- Installer globalement ce module : https://www.npmjs.com/package/nodemon

Nodemon est un outil qui permet de développer des applications basées sur Node.js en redémarrant automatiquement l'application lorsque des modifications de fichiers dans le répertoire sont détectées (hors variables d'environnements).

- Ajouter un script « start » qui prend en valeur la commande « nodemon server ».

- Lancer votre serveur avec la commande définie.

ACTIVITE 3 : Hello World avec Node.js

Dans le fichier server.js précédemment créé, ajouter le code suivant :

```
//Import du module http de Node.js qui permet de créer des serveurs web HTTP
const http = require('node:http');


//On définit l'adresse ip sur laquelle le serveur va s'éxécuter
const hostname = '127.0.0.1'
//On définit le port sur lequel le serveur va écouter.
const port = 3000;


//On crée un serveur HTTP avec la méthode createServer() du module http.
const server = http.createServer((req, res) => {
  //On définit la réponse à renvoyer au client.
  res.statusCode = 200; //Choix du code HTTP
  res.setHeader('Content-Type', 'text/plain'); //Choix du type de contenu
  res.end('Hello, World!\n'); //Envoi du contenu de la réponse au client
});


//On demande au serveur d'écouter les connexions entrantes sur le port et l'adresse ip spécifiée.
server.listen(port, hostname, () => {
  //On affiche un message en console avec le lien d'accès au serveur
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Vous devriez voir le message « Server running at http://127.0.0.1:3000/ » dans le terminal et « Hello, World! » en ouvrant l'url dans le navigateur.

6. npx

npx est un outil installé automatiquement avec Node.js. C'est l'acronyme de Node Package eXecute. Il permet d'exécuter des commandes en ligne de commande sans installer les packages correspondants globalement ou de lancer des commandes spécifiques à un projet à partir de packages installés localement.

7. nvm

nvm ou Node Version Manager est un outil en ligne de commande qui permet de gérer les différentes installation de Node.js sur un même système. Il permet d'installer et de gérer plusieurs versions. On peut alors facilement passer de l'une à l'autre selon le projet sur lequel on travaille.

Installation sur Linux/MacOS :

- https://github.com/nvm-sh/nvm#installing-and-updating
- https://github.com/nvm-sh/nvm#usage

Installation sur Windows :

- https://github.com/coreybutler/nvm-windows#installation--upgrades
- https://github.com/coreybutler/nvm-windows#usage

IV. Les APIs REST

A. Qu'est-ce qu'une API ?

Aujourd'hui nous avons de plus en plus de supports différents : ordinateur, téléphone, tablette, télévision, logiciels embarqués dans les voitures, etc. Comment faire communiquer toutes ces applications ? Comment Netflix ou YouTube synchronisent-ils tous ces supports ? **Ils utilisent des APIs !**

Définition de la CNIL : *« Une API (application programming interface ou « interface de programmation d'application ») est une interface logicielle qui permet de « connecter » un logiciel ou un service à un autre logiciel ou service afin d'échanger des données et des fonctionnalités.*

Les API offrent de nombreuses possibilités, comme la portabilité des données, la mise en place de campagnes de courriels publicitaires, des programmes d'affiliation, l'intégration de fonctionnalités d'un site sur un autre ou l'open data. Elles peuvent être gratuites ou payantes. »

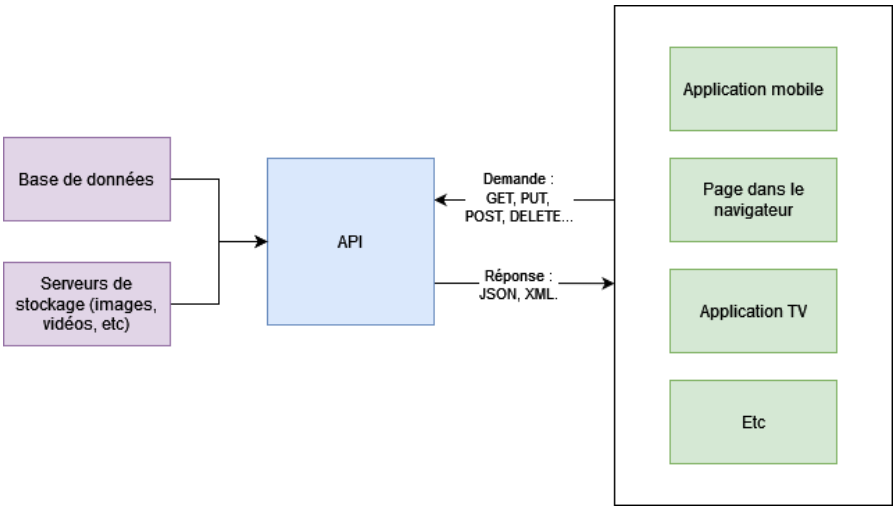


Figure 3: Schématisation du fonctionnement d'une API

B. De l'API à l'API REST

L'API REST ou RESTful est un type d'API qui respecte l'architecture REST. REST est l'acronyme de « REpresentational State Transfer ». C'est un style d'architecture dédiée à la création d'APIs. Ce style d'architecture a été proposé par Roy Fielding lors de sa thèse de doctorat en 2000.

On ne parle pas ici de différence de protocole ou de norme mais bien de contraintes à respecter en terme d'architecture. Les six principales contraintes sont :

- **L'architecture client-serveur** : on garantit la séparation des responsabilités. Le client s'occupe des problèmes d'interface avec l'utilisateur tandis que le serveur s'occupe des problèmes de stockage des données. L'API REST peut alors servir plusieurs clients sans se préoccuper de l'aspect des interfaces et de ce qu'elles font.
- **L'absence d'état ou « stateless »** : les informations du client ne sont jamais stockées entre les requêtes. Chaque demande doit être autonome et complète.
- **La mise en cache** : on stocke les réponses de l'API qui ne changeront pas ou peu sur une certaine période. On bloquera en revanche la mise en cache des réponses qui changent constamment. On gagne alors en performance.
- **Le système par couche** : invisible pour le client, il permet de hiérarchiser les différents types de serveurs (balancing, sécurité…) utilisés pour récupérer les informations. On gagne en évolutivité, sécurité et flexibilité.
- **L'interface standardisée** :
    - On utilise une URL pour envoyer la requête, cette URL spécifie la ressource demandée. C'est ce qu'on appelle l'identification des ressources.
    - L'API renvoi une représentation de la ressource dont le format peut différer de celui présent sur le serveur (ex : données stockées en sql renvoyées en json).
    - Emission de messages autodescriptifs au client suffisamment détaillés pour que celui-ci sachent comment traiter les informations. (ex : en-tête json)
    - Présence d'hypermédia rendant l'API auto découvrable (ex : URL pour la modification, la suppression…)
- **Le code à la demande** (facultatif) : on peut envoyer du code exécutable du serveur vers le client si ce dernier le demande.

C. Le modèle de Richardson

C'est un modèle qui décrit les différentes étapes de maturité lors de la conception d'une API REST. Il a été proposé par Leonard Richardson en 2008 et est bien sûr basé sur les travaux de Roy Fielding. Il est formé de quatre niveaux :

- Niveau 0 : On se contente d'utiliser HTTP comme canal de transport. L'API peut être contactée via un point d'entrée unique (une seule URI). Les messages échangés sont du POX (Plain Old XML). Dans tous les cas, un code 200 est renvoyé avec des informations sur le bon ou mauvais déroulement de l'action. On est donc pas encore REST à ce niveau.
- Niveau 1 : on inclut ici la différenciation des ressources. Ce qui signifie qu'on a des ressources individuelles et non pas un unique point d'entrée.
- Niveau 2 : on ajoute la gestion des verbes (GET, POST…) et codes retour (200, 404, 301…) HTTP.
- Niveau 3 : on ajoute les liens hypermédias (HATEOAS). Pour chaque ressource un élément « link » fournissant les URI permettant de la manipuler sont fournis.

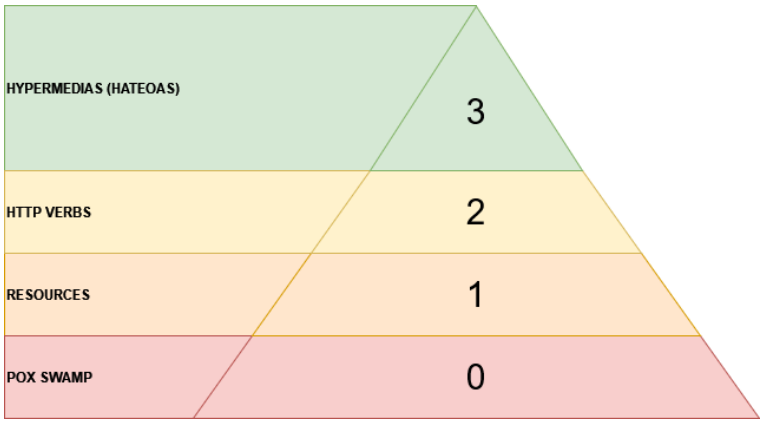


Figure 4: Représentation visuelle du modèle de Richardson

D. Verbes HTTP

Les principaux verbes HTTP sont :

- POST ➔ Créer une nouvelle ressource.
- PUT ➔ Mettre à jour une ressource existante.
- GET ➔ Récupérer des données.
- DELETE ➔ Supprimer une ressource.
- PATCH ➔ Mettre à jour partiellement une ressource. On peut alors n'inclure que les données à mettre à jour dans le corps de la requête plutôt que de remplacer complètement la ressource.

Pour aller plus loin : Tous les verbes HTTP

https://developer.mozilla.org/fr/docs/Web/HTTP/Methods

E. Codes HTTP

Il y a globalement cinq catégories de codes :

- 1xx ➔ Information.
- 2xx ➔ Succès.
- 3xx ➔ Redirection.
- 4xx ➔ Erreur du client.
- 5xx ➔ Erreur du serveur.

Voici les principaux codes à connaitre :

- 200 ➔ OK : requête traitée avec succès.
- 201 ➔ Created : requête traitée avec succès et création d'un document.
- 204 ➔ No content : requête traitée avec succès mais pas d'information à renvoyer.
- 301 ➔ Moved Permanently : Document déplacé de façon permanente.
- 304 ➔ Not modified : document non modifié depuis la dernière requête.
- 401 ➔ Unauthorized : une authentification est nécessaire pour accéder à la ressource.
- 403 ➔ Forbidden : l'accès est refusé et s'identifier ne changera rien. Il manque les droits d'accès.
- 404 ➔ Not found ➔ Ressource introuvable.

Pour aller plus loin : Tous les codes HTTP

https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

V. Une application Node.js avec Express

Express est un framework web minimaliste pour Node.js, utilisé pour créer des applications web côté serveur. Il fournit des fonctionnalités pour la gestion des routes, des middlewares, des requêtes, des réponses, des sessions, et bien plus encore. Express est très populaire en raison de sa simplicité et de sa flexibilité, ce qui permet aux développeurs de créer rapidement des applications web de haute qualité.

A. Création du serveur

ACTIVITE 4 : Création d'un serveur Node.js avec Express

- Installer express : https://www.npmjs.com/package/express

- Réaliser le « Hello world » : https://expressjs.com/fr/starter/hello-world.html

Vous devriez voir le message « Example app listening on port 3000» dans le terminal et « Hello World! » en ouvrant l'url dans le navigateur (http://localhost:3000).

Rappel : les imports/exports en JS

Comme JavaScript, Node.js permet l'utilisation de modules CommonJS (avec « require ») ou de modules ECMAScript avec « import … from… ».

Les modules CommonJS : https://nodejs.org/api/modules.html

Les modules ECMAScript : https://nodejs.org/api/esm.html

B. Variables d'environnement

Lors de l'exercice précédent, nous avons défini le port 3000 en sein du fichier server.js. Il serait plus pratique de centraliser toutes nos variables d'environnement dans un seul fichier.

Voici une liste non exhaustives des variables d'environnement :

- La configuration du serveur : PORT, HOST…
- La base de données : DATABASE_URI, DATABASE_HOST, DATABASE_USER…
- L'outil d'envoi de mail : MAILER_HOST, MAILER_PASSWORD…
- Les tokens : JWT_TOKEN, PASSPHRASE, IV…
- La connexion aux APIs : MAILCHIMP_TOKEN, STRIPE_TOKEN…
- Etc

ACTIVITE 5 : Mise en place de la variable d'environnement pour le port

- Installer dotenv : https://www.npmjs.com/package/dotenv

- Créer un fichier .env dans lequel on insère la variable `PORT=3000`.

- Dans le fichier server.js, importer dotenv : `require('dotenv').config()` puis remplacer la valeur « 3000 » par la variable d'environnement (process.env.PORT). Redémarrer le serveur.

- Essayer de changer la variable port par 5000, redémarrer le serveur (nodemon ne peux pas recompiler l'environnement) et vérifier que le serveur tourne maintenant bien sur le port 5000.

Les données dans le fichier .env sont très sensibles. On ne les envoi alors jamais sur GitHub. Il peut en revanche être utile d'avoir l'architecture de ce fichier quand on récupère un projet pour savoir quoi remplir. On crée alors en général un fichier .env.example avec des données vides, qui lui, va être présent sur GitHub.

Exemple :

```
MONGO_URI=

JWT_TOKEN=
```

ACTIVITE 6 : Fichier .env.example

Créer le fichier .env.example adapté à votre projet.

C. Architecture du projet

Voici une structure possible à adopter :

- App
  - config
  - controllers
  - middleware
  - models
  - routes
- Les autres fichiers à la racine (package.json, .env, .gitignore, server.js…)

ACTIVITE 7 : Séparer la logique de l'application de la logique du serveur

- Créer un fichier app.js à la racine du projet.

- Y insérer le code suivant :

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

const app = express()

module.exports = app;
```

- Retirer alors ce code du fichier server.js et importer le fichier app.js à la place :

```
require('dotenv').config()

const app = require("./app.js");

const port = process.env.PORT
```

```
app.get('/', (req, res) => {
  res.send('Hello World!')
})


app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

D. Connexion à la base de données avec Sequelize

Sequelize est un ORM (Object Relational Mapping) pour Node.js. Il facilite les interactions avec les bases de données relationnelles (MySQL, MariaDB, PostgreSQL…). On peut alors définir facilement des modèles de données en JS et effectuer des opérations CRUD (Create, Read, Update et Delete). On peut aussi créer des jeux de données grâce à *sequelize-cli* et gérer les migrations.

ACTIVITE 8 : Mise en place de Sequelize et des modèles

INSTALLATION

- Installer sequelize : https://sequelize.org/docs/v6/getting-started/#installing (juste la partie Installing).

- Installer sequelize-cli : https://sequelize.org/docs/v6/other-topics/migrations/#installing-the-cli (juste la partie Installing the CLI)

INITIALISATION

- Se placer en commande dans le dossier app.

- Initialiser sequelize : `$ npx sequelize-cli init`

- Se rendre dans config/config.json et ajuster la configuration. Créer manuellement les bases de données sur phpmyadmin.

- Se rendre dans le fichier app.js à la racine du projet et ajouter :

```
…
const db = require("./app/models/index.js");
db.sequelize
  .authenticate()
  .then(() => console.log("Database connected ..."))
  .catch((err) => console.log(err));
…
```

Ceci permet d'essayer de s'authentifier auprès de la base de données dès le démarrage du serveur et de voir ainsi directement s'il y une erreur.

CREATION DES MODELES

Ressources utiles pour les modèles : Model basics | Other data types | Validations and constraints

- Création du modèle user : `$ npx sequelize-cli model:generate --name User --attributes firstName:string,lastName:string,email:string,password:string`

Ajuster le modèle de sorte :

- Qu'aucun champs ne puisse être nul (avec un message personnalisé).

- Que l'email soit unique et qu'il soit au bon format.

- Création du modèle wood : `$ npx sequelize-cli model:generate --name Wood --attributes name:string,type:enum,hardness:enum`

Ajuster le modèle de sorte QUE :

- le « type » en ENUM puisse prendre les valeurs "softwood", "exotic wood", "noble and hardwoods".

- le « hardness » en ENUM puisse prendre les valeurs "tender", "medium-hard", "hard"

Pensez également à mettre à jour les fichiers de migrations.

MIGRATIONS

- Créer une nouvelle migration (non nécessaire ici) : https://sequelize.org/docs/v6/other-topics/migrations/#migration-skeleton

- Effectuer la migration : `$ npx sequelize-cli db:migrate`

https://sequelize.org/docs/v6/other-topics/migrations/#running-migrations

Vous devriez à présent voir vos tables sur phpmyadmin

JEU DE DONNEES OU « SEEDS »

- Créer un jeu de données pour user (demo-user) : `$ npx sequelize-cli seed:generate --name demo-user`

Ceci génère la structure d'un jeu de données vide dans lequel il faut ajouter les données, par exemple :

```
'use strict';

/** @type {import('sequelize-cli').Migration} */

module.exports = {

  async up (queryInterface, Sequelize) {

    await queryInterface.bulkInsert('Users', [{

      firstname: 'John',

      lastname: 'Doe',

      email: "john.doe@mail.com",
```

```
        password: "123456",
      }], {});
  },

  async down (queryInterface, Sequelize) {
    await queryInterface.bulkDelete('Users', null, {});
  }
};
```

- Faire de même pour wood (demo-wood)

- Lancer la commande `$ npx sequelize-cli db:seed:all` pour envoyer les données dans la base.

Vous devriez à présent avoir vos données en BDD.

E. Créer les routes

Liens qui vous seront utiles tout au long de cette partie du cours :

- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes
- https://expressjs.com/fr/guide/routing.html

ACTIVITE 9 : Les routes

- Retirer la route « / » sur fichier server.js.

- Créer un fichier index.js dans le dossier « routes ».

- Nous allons ensuite créer nos routes en trois temps :

1. Création de la route « /api » qui renvoie vers le fichier index.js du dossier « routes ». Dans le fichier app.js :

```
const router = require("./app/routes/index.js");

//Ajout des routes

app.use("/api", router);
```

2. Création des groupes de routes dans le fichier index.js du dossier « routes ». Nous allons créer un groupe pour les routes concernant l'utilisateur et un groupe pour les routes concernant les essences de bois. Dans le fichier routes/index.js (exemple uniquement pour l'utilisateur, à vous de le reproduire pour les essences de bois):

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

const router = express();

const userRoutes = require('./user.js') //Ce fichier n'existe pas encore.


router.use("/auth", userRoutes)


module.exports = router
```

3. Création des routes finales dans les fichiers routes/user.js et routes/wood.js dédiées. Exemple pour l'utilisateur :

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


router.post('/signup', function (req, res) {
    res.send('You are signup');
  });
router.post('/login', function (req, res) {
    res.send('You are login');
  });


module.exports = router;
```

Pour les essences de bois, vous ajouterez seulement une route en GET avec le message « List of woods ».

Outil important : Postman

Postman est un logiciel qui permet d'envoyer des requêtes vers une API. On peut ainsi tester l'ensemble des routes et des réponses (succès et erreur).

Téléchargement : https://www.postman.com/downloads/

Documentation : https://learning.postman.com/docs/introduction/overview/

F. Création des contrôleurs

Lors de l'activité précédente, nous avons créé des routes qui ne renvoi que de simples messages. Les contrôleurs vont nous permettre traiter et renvoyer les données. L'objectif est ici de rendre nos routes fonctionnelles.

1. Contrôleurs basiques

ACTIVITE 10 : Créer les contrôleurs

Dans cette première activité, nous allons renvoyer les mêmes messages que précédemment mais en appelant cette fois des méthodes dans des contrôleurs.

- Créer les fichiers controllers/user.js et controllers/wood.js

- Exemple pour le « signup » :

- Fichier controllers/user.js :

```
exports.signup = (req, res) => {
      res.send('You are signup');
};
```

*« req » représente l'objet « **req**uest » envoyé par le client et qui contient les informations sur la demande http (url, paramètres, en-têtes, données…).*

*« res » représente l'objet « **res**ponse » qui sera renvoyé au client avec les données demandées ou une erreur le cas échéant.*

-Fichier routes/user.js

```
const express = require("express");

const router = express();

const userCtrl = require("../controllers/user.js");


router.post("/signup", userCtrl.signup);

//…


module.exports = router;
```

- A vous de jouer pour les deux autres routes !

2. Récupérer des données

On va avoir besoin d'un outil pour transformer les données envoyées dans la demande sous format JSON en objet javascript manipulable. On va alors utiliser la méthode express.json().

ACTIVITE 11 : Ajout de la méthode express.json()

Dans le fichier app.js :

```
//…

app.use(express.json());

//Ajout des routes

//..
```

ACTIVITE 12 : Rendre l'inscription fonctionnelle

Dans cette activité nous allons récupérer les données envoyées par le client et ainsi enregistrer notre nouvel utilisateur en base de données. Nous allons alors travailler dans le fichier controllers/user.js et dans la méthode signup. Voici les étapes à suivre :

- Récupérer le corps de la requêtes avec « req.body » : https://expressjs.com/en/4x/api.html#req.body

- Utiliser la méthode « create » de sequelize pour créer l'utilisateur en BDD : https://sequelize.org/docs/v6/core-concepts/model-instances/#a-very-useful-shortcut-the-create-method. Vous penserez à importer votre model User.

- En cas de succès vous renverrez l'utilisateur, sinon vous renverrez l'erreur levée.

ACTIVITE 13 : Encrypter le mot de passe

Vous utiliserez le module bcrypt pour hacher le mot de passe avant l'enregistrement en base de données. (Pensez à mettre à jour vos « seeds »).

3. Renvoyer des données

ACTIVITE 14 : Récupérer la liste des types de bois

L'objectif est ici de renvoyer la liste des différents type de bois. Voici les étapes à suivre :

- Utiliser la méthode « findAll » de sequelize : https://sequelize.org/docs/v6/core-concepts/model-querying-basics/#simple-select-queries. Vous penserez à bien importer le modèle Wood.

- Vous renverrez tous les résultats sous format json.

ACTIVITE 15 : Récupérer la liste des bois selon leur dureté

Nous allons maintenant créer une nouvelle route et une nouvelle méthode de contrôleurs nous permettant de récupérer les essences de bois selon leur dureté. Voici les étapes :

- Créer la nouvelle route dans routes/wood.js. Cette route devra prendre un argument « :hardness » soit la route « /:hardness ». Vous pourrez ensuite récupérer ce paramètre dans votre contrôleur en utilisant « req.params.hardness ».

- Créer la nouvelle méthode findByHardness dans le controllers/wood.js.

- Utiliser la méthode « findAll » de sequelize avec une clause « where » pour ne récupérer que les bonnes essences de bois en fonction de la dureté précisée dans l'URL : https://sequelize.org/docs/v6/core-concepts/model-querying-basics/#applying-where-clauses

G. Les middlewares

Un middleware est une fonction placée entre la requête d'un client et la réponse qui lui est apportée. Il intercepte la requête et effectue un certain nombre d'action : modification de la requête, vérification de l'authentification du client, validation de données, upload de fichier… Les middlewares sont généralement chainés. Lorsque le premier a terminé son travail, il passe la main au suivant et ce jusqu'à la réponse finale.

L'intérêt du middleware est double :

- Il permet de définir des fonctions réutilisables facilement sur différentes routes de l'application.
- Il permet la séparation des préoccupations, chaque middleware se concentre sur une tâche spécifique.

ACTIVITE 16 : Rendre la connexion fonctionnelle avec JWT

Ici nous souhaitons pouvoir authentifier nos utilisateurs. Nous allons pour cela utiliser un **jsonwebtoken**. La marche à suivre est la suivante dans la méthode login de controllers/user.js :

- Retrouver l'utilisateur grâce à son adresse e-mail (req.body.email) et à la fonction findOne de sequelize : https://sequelize.org/docs/v6/core-concepts/model-querying-finders/#findone (et la clause « where » bien sûr).

- Si l'utilisateur n'est pas trouvé, renvoyer une erreur, sinon comparer le mot de passe fourni par l'utilisateur avec la fonction compare de bcrypt.

- Si le mot de passe est correct, construire le token et le renvoyer avec les données de l'utilisateur.

ACTIVITE 17 : Rendre notre API Privée

Nous aimerions maintenant que seul les utilisateurs authentifiés puissent accéder à la liste des essences de bois. On va alors créer notre premier middleware. Nous allons donc :

- Créer un dossier middleware dans le dossier app.

- Dans ce dossier créer un fichier auth.js.

- Y insérer le code suivant :

```
const jwt = require('jsonwebtoken');


module.exports = (req, res, next) => {
  try {
    const token = req.headers.authorization.split(' ')[1];
    //decoded the token
    const decodedToken = jwt.verify(token, process.env.TOKEN_SECRET);
    const userId = decodedToken.id;
   //save userId in req.auth var
    req.auth = {
      userId
    };
    next();
  } catch (err) {
    res.status(401).json({
      error: 'Unauthorized request!'
    });
  }
};
```

- Ajouter le middleware aux routes à verrouiller (routes/wood.js) :

```
//…
const auth = require("../middleware/auth.js")


router.get("/", auth, woodCtrl.readAll);
router.get("/:hardness", auth, woodCtrl.readByHardness);
//…
```

On ne verrouille pas les routes login et signup puisqu'en effet si on y accède c'est bien qu'on n'est pas connecté.

ACTIVITE 18 : Ajouter une image aux essences de bois

Nous aimerions maintenant pouvoir illustrer chaque essence de bois par une photo. On va donc devoir :

- **Ajouter un champs « image » de type string qui peut être nul dans le modèle woods** ainsi que dans nos migrations et mettre à jour notre base de données. On ne stockera en effet que le lien vers l'image sur notre serveur et non pas l'image en elle-même.

- **Créer la route et la méthode de contrôleur** de création d'une nouvelle essence de bois.

- **Installer le middleware** multer : https://www.npmjs.com/package/multer

- **Surcharger le middleware** en créant un nouveau fichier multer.js dans le dossier middleware :

```
const multer = require("multer");
const fs = require('fs');


//On définit les extensions selon le mime type.
const MIME_TYPES = {
    "image/jpg" : "jpg",
    "image/jpeg" : "jpg",
    "image/gif" : "gif",
    "image/png" : "png",
    "image/webp" : "webp",
};


//On crée le dossier uploads s'il n'existe pas
if (!fs.existsSync('uploads')) {
    fs.mkdirSync('uploads');
}
```

```
// diskStorage => destination du fichier / générer un nom de fichier unique
const storage = multer.diskStorage({
  destination: (req, file, callback) => {
    callback(null, "uploads");
  },
  filename: (req, file, callback) => {
    const name = file.originalname.split(" ").join("_").split(".")[0]
    const extension = MIME_TYPES[file.mimetype]
    callback(null, name +  "_" + Date.now() + "." + extension);
  },
});

// On exporte le module avec ces paramètres en précisant
// qu'on attend un champ "image"
module.exports = multer({storage: storage}).single("image");
```

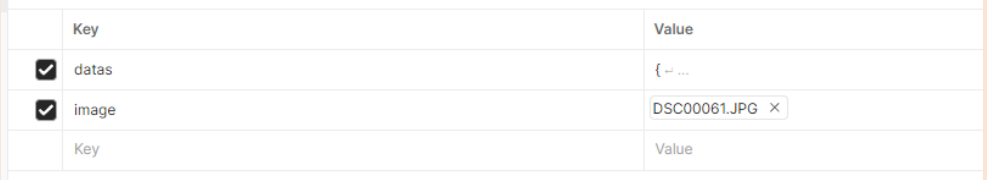
- **Servir statiquement les assets** : actuellement votre API n'a qu'un seul point d'entrée la route « /api » suivie des autres paramètres de route. Nous aimerions maintenant pouvoir servir les images sans avoir à récréer un contrôleur, à retrouver l'image et la renvoyer. On peut faire ceci en utilisant **express.static()**. On va alors ajouter à notre fichier app.js :

```
//…
app.use("/uploads", express.static(path.join(__dirname, "uploads")));
//…
```

- **Utilisons maintenant notre middleware** au niveau de la route de création d'une essence de bois :

```
//…
const multer = require('../middleware/multer.js')
//…
router.post("/", auth, multer, woodCtrl.create);
//…
```

- **Adapter la méthode du contrôleur** : vous avez dans un premier temps utilisé req.body pour récupérer les données de création de l'essence de bois. Or, maintenant qu'on utilise aussi une image, nous allons utiliser un type de formulaire particulier : le **form-data**. Vous aurez donc deux clés : req.file pour ce qui est relatif au fichier et req.body.*key*_name pour la clé définie avec le contenu de la requête (ici « datas »).



Dans le contrôleur il va falloir :

- **Construire dynamiquement le chemin de l'image** en utilisant req.protocol et req.get("host") suivi du nom du dossier « uploads » suivi du nom de l'image :

```
const pathname = `${req.protocol}://${req.get("host")}/uploads/${req.file.filename}`;
```

- **Adapter l'objet** renvoyé lors du « create » :

```
{
  ...JSON.parse(req.body.datas), //Transforme les données en format utilisable
  image: pathname,
}
```

Vous devriez à présent pouvoir créer et récupérer les essences de bois et voir les images grâce à leur URL.

Pour aller plus loin : le système de fichier de Node.js

https://nodejs.org/api/fs.html

H. Gestion des cors

La gestion des CORS ou « Cross-Origin Resource Sharing » en Node.js se réfère à la façon dont les requêtes http provenant de différentes origines sont adoptées ou non par le serveur. C'est une mesure de sécurité qui permet de contrôler l'origine des requêtes. On peut par exemple, n'accepter que les requêtes provenant d'un nom de domaine précis.

Nativement, si on ne précise rien le serveur rejettera toutes les requêtes dont le nom de domaine est différent du sien.

On utilisera alors généralement un module npm comme « cors » ou « helmet ».

ACTIVITE 19 : Autoriser uniquement les requêtes provenant d'un domaine

En utilisant le module npm « cors », autoriser uniquement les requêtes provenant du domaine http://localhost:8080.

VI. Liste de modules utiles

- sequelize ➔ Communiquer avec une base de données relationnelle.
- mongoose ➔ Communiquer avec une base de données non relationnelle (MongoDB).
- crypto-js ➔ Encrypter/Décrypter des données.
- helmet ➔ Bibliothèque de middleware concernant la sécurité (en-têtes de sécurité).
- express-rate-limit ➔ Pour limiter le nombre de requête par IP ou clé API.
- express-slow-down ➔ Pour ralentir les requêtes à partir d'un certain nombre de requêtes provenant d'une IP ou clé API.
- bcrypt ➔ hashage des mots de passe.
- express-hateoas-links ➔ Pour générer les liens HATEOAS (hypermedia).
- jsonwebtoken ➔ Pour créer et vérifier des tokens (jetons d'authentification).

Pour aller plus loin : la sécurité et Node.js

https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html

VII. Debug, analyse et performance

A. Outils de debug

1. Le debugger Node.js

Il permet de placer des points d'arrêt dans le code et de pouvoir le parcourir pas à pas pour en comprendre son fonctionnement. On peut inspecter les variables et le comportement de l'application.

https://nodejs.org/api/debugger.html

2. L'inspecteur Node.js

Inspect est un autre outil qui permet d'inspecter le code Node.js en utilisant la CLI. On peut l'utiliser pour lancer l'API en mode débogage et en attachant un outils de débogage tiers comme Google chrome (ou les navigateurs basés sur Chrome).

https://nodejs.org/api/inspector.html

3. La journalisation

Il est très intéressant de mettre en place un système de logs sur l'API afin de pouvoir voir les erreurs ou avertissements levés sur une période donnée. En effet, vous ne serez pas toujours derrière votre ordinateur à vérifier que chaque requête du client se déroule bien. Quelques modules réputés :

- Winston ➔ un des modules de journalisation les plus personnalisables. On peut facilement choisir les informations et destinations des messages.
- Pino ➔ réputé pour sa rapidité et facilité d'utilisation.
- Morgan ➔ C'est un middleware express qui permet la journalisation des requêtes http.

B. Outils d'analyse du code

1. ESLint

ESLint est un outil d'analyse statique du code JS. On l'utilise pour détecter les problèmes de qualité de code, les erreurs de syntaxe, de style, de logique, de sécurité… Il analyse le code et signale les erreurs ou avertissements en se basant sur des règles prédéfinies ou personnalisées.

On peut l'intégrer facilement via un module *npm*, via l'IDE ou encore lors de l'intégration continue.

https://eslint.org/docs/latest/use/getting-started
https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint

2. DeepScan

DeepScan est un outil similaire à CodeClimate mais pour JavaScript. Il analyse la qualité du code, les conventions, la sémantique, etc. Il est très facilement connectable à un repository GitHub et on peut alors le lancer automatiquement à chaque pull request (intégration continue).

https://deepscan.io/

3. Snyk

Snyk est un peu différent car il va lui analyser les vulnérabilités des modules *npm*. Dès lors qu'il en détectera une, il créera automatiquement une pull request sur votre repository que vous n'aurez alors plus qu'à merger. Bien sûr différentes configurations sont possibles.

https://snyk.io/fr/

VIII. Déployer une API Node.js

Il existe plusieurs plateforme pour déployer un projet Node.js :

- Heroku
- Cyclic ➔ **offre gratuite**
- AWS Elastic Beanstalk
- Google cloud Platform
- Microsoft azure
- Etc

On peut généralement connecter le repository GitHub et associer une branche à automatiquement déployer en production. On peut aussi configurer par exemple des commandes ou encore les variables d'environnement.

IX. Autres frameworks et langages

Nous avons tout au long de ce cours travaillé avec le framework Express.js qui est sans doute le plus populaire. Sachez qu'il existe d'autres frameworks basés sur Node.js, comme :

- NestJS ➔ Un framework qui suit les principes SOLID ce qui lui permet d'offrir flexibilité et évolutivité.
- KoaJS ➔ Un framework moderne et dominant. Souvent décrit comme une version légère d'express.
- Sails.js ➔ Un framework proche du modèle MVC. Très adapté aux applications web en temps réel (ex : chat).
- Hapi.js ➔ Un framework qui insiste sur la sécurité et la configuration.

Sachez également qu'il est tout à fait possible d'utiliser Node.js avec TypeScript :
https://nodejs.dev/fr/learn/nodejs-with-typescript/