

# NodeJS

Support de cours IUT de Calais

Kevin GUERRIER

## Préambule

Lien vers le support de cours

<https://bit.ly/2024-iut-nodejs-cm-kguerrier>

*Ce support est fortement inspiré du support de Valentin FOURNY, intervenant également à l'IUT de Calais et qui a mis à disposition sur*

<https://formation-vfourny.vercel.app/node>

*Pour la partie pratique, à partir de github Classroom :*

<https://bit.ly/2025-iut-nodejs-classroom-kguerrier>

*Si le lien ci-dessus ne fonctionne pas : <https://classroom.github.com/a/z69Mbtqf>*

*Et bien évidemment le lien vers*

*la documentation officielle : <https://nodejs.org/>*

## Au commencement...

Node.js est une technologie apparue en 2009, créée par Ryan DAHL. À cette époque, le développement web reposait principalement sur des serveurs classiques comme Apache. Pour bien aborder le fonctionnement de NodeJS, il est donc utile de faire un retour sur le fonctionnement d'un site web.

Lorsque vous accédez à un site web via votre navigateur vous utilisez une url. Basiquement, cette url permet une communication entre 2 machines principales : la vôtre (c'est le poste client) et celle où le site est hébergé (c'est le serveur).

## Côté client (FRONTEND)

Le client c'est vous. Ce qui est chargé du côté de votre navigateur, et cela concerne plus généralement l'ensemble des interactions que vous avez avec la page. Les langages dit serveur (comme PHP, Ruby) ne traitent pas les informations de ce côté, c'est plutôt du ressort des langages client comme Javascript.

## Côté serveur (BACKEND)

Les ordinateurs côté serveurs sont différents des ordinateurs que les clients utilisent. Historiquement, il s'agit d'ordinateurs assez puissants, qui tournent nuit et jour et ne servent globalement qu'à stocker et faire tourner des services / des applications sur le web. Dans le cas d'une application web, elle va appeler, via des requêtes API, différents scripts qui vont s'exécuter de ce côté. Il peut s'agir d'appels à une base de données, de traitements reçus par le client à vérifier ou alors de données à lui renvoyer. C'est la partie qui nous intéresse.

## Fonctionnement

Contrairement à un site statique où chaque page est envoyée en totalité au client, sans modification de la page en temps réel, le site dynamique va permettre une certaine interaction. En effet, lorsque le client va exécuter des actions précises, comme par exemple valider un formulaire d'inscription, ces informations de connexion vont être envoyées au serveur, être vérifiées et une réponse va ensuite être renvoyée côté client pour faire un affichage différent, par exemple, en fonction du cas d'une connexion réussie ou non.

## Et NodeJS dans tout ça

Javascript permet à la base un langage de script prévu pour apporter des fonctionnalités côté client dans le but de faire du web dynamique. NodeJS est une bibliothèque se basant sur javascript mais qui offre la possibilité de gérer un backend.

En d'autres termes, il est interprété côté serveur et non plus côté client (navigateur). C'est d'ailleurs son rôle principal : gérer des interactions backend (côté serveur). Et l'une de ses spécificités est notamment qu'il permet de très bien gérer les requêtes multiples asynchrones (qui permettent de poursuivre leur exécution en attendant des réponses et en exécutant certaines parties de code dès que la réponse est disponible).

Bien que Node possède ses propres caractéristiques, il peut (dans une certaine mesure) être comparé à des langages tels que PHP ou Ruby.

## Principales différences

Caractéristique	JavaScript (dans un navigateur)	Node.js
Contexte d'exécution	Navigateur (Chrome, Firefox, etc.)	Côté serveur
Modules intégrés	Aucune, sauf via des APIs du DOM	Oui (fs, http, path, etc.)
Gestion des fichiers	Non	Oui (via le module fs)
Modules tiers	Non	Oui, via npm

<b>Utilisation principale</b>	Interaction utilisateur, DOM, animations	Backend, API, tâches système
-------------------------------	--	------------------------------

## NodeJS vs les autres

Les plates-formes web traditionnelles utilisent en général un système de multi-threading (plusieurs thread en parallèle) pour gérer la concurrence. C'est-à-dire que pour chaque requête transmise au serveur, un thread se charge de la gérer, et il n'est libéré qu'une fois que les opérations demandées par la requête en question sont achevées. Le thread est donc bloqué durant ce fonctionnement.

Une grande partie des opérations web sont des appels à la base de données ou des appels à une api externe, ce qui est donc relativement consommateur en temps et en ressources. Et le serveur se retrouve donc dans l'obligation de créer autant de thread que de connexion.

Dans un contexte où le nombre de requêtes peut rapidement grimper (des milliers de connexions en simultané), le multi-threading atteint rapidement ses limites. En effet, il faut savoir que le serveur ne peut dépasser un certain nombre de thread simultanément, et il y a donc un moment où, faute de thread disponible, les nouvelles requêtes ne pourront plus être prise en charge, et se verront donc être mises en attente, ce qui impactera drastiquement les performance de l'application.

C'est là que Node.js prend un fonctionnement différent car il est single-threaded. C'est-à-dire qu'un seul et unique thread peut tourner à la fois !

Cela peut sembler inefficent à première vue compte tenu de l'infrastructure multi-coeurs que nous avons à disposition aujourd'hui. Mais ce qu'il faut savoir, c'est qu'en contrepartie de ce fonctionnement sur le même thread, la plupart des opérations sont non bloquantes.

C'est donc par cette nature asynchrone que le besoin d'attente est éliminé, permettant ainsi au serveur de gérer des milliers et des milliers de connexions simultanées, sans que cela impacte les performances de l'application.

Au fil des années, Node.js a connu plusieurs évolutions majeures.

Dès 2011, il gagne en popularité avec l'apparition de npm, son gestionnaire de paquets, qui facilite l'intégration de bibliothèques tierces.

En 2015, une scission au sein de la communauté entraîne la création d'io.js, qui sera finalement réintégré à Node.js sous la gouvernance de la Node.js Foundation.

Depuis, Node.js continue d'évoluer avec des améliorations sur la gestion des threads (worker threads), le support des modules ECMAScript (ESM) et l'optimisation de la gestion mémoire et des performances.

# Une autre notion importante de NodeJS : les APIs

## Qu'est-ce qu'une API ?

Une API (Application Programming Interface, ou interface de programmation d'applications) est un ensemble de règles et de conventions qui permet à des applications logicielles de communiquer entre elles. Elle agit comme un intermédiaire, facilitant l'échange d'informations ou de services entre différents systèmes.

## Les composantes clés d'une API :

- L'Interface
  - Une API définit comment une application peut interagir avec une autre, souvent via des points d'accès (ou endpoints).
  - Exemple : Une application web peut utiliser une API pour demander des données d'un serveur.
- Les notions de Requête et de Réponse
  - Les interactions avec une API impliquent des requêtes envoyées par un client et des réponses fournies par le serveur.
  - Exemple : Une requête GET à une API peut récupérer des données, tandis qu'une requête POST peut envoyer des informations.
- Le Protocole de communication
  - Les API modernes utilisent souvent le protocole HTTP/HTTPS.
- Le Format des données
  - Les API utilisent des formats standardisés pour échanger des données, comme :
    - JSON (JavaScript Object Notation).
    - XML (Extensible Markup Language).

## Quelques types d'API

- L'API web
  - Elle permet à une application d'accéder à des fonctionnalités ou des données via Internet.
  - Exemple : L'API de Google Maps fournit des données de cartographie.
- L'API système
  - Elle fournit un accès aux fonctionnalités du système d'exploitation.
  - Exemple : Une API système peut permettre d'accéder au système de fichiers.
- L'API de bibliothèque/framework
  - Elle fournit des fonctionnalités prêtes à l'emploi dans des bibliothèques ou des frameworks.
  - Exemple : L'API DOM pour manipuler des éléments HTML.

# Les bases de NodeJS

Le but de cette section est de comprendre les bases de NodeJS et comment démarrer un projet.

## Installation de NodeJS

Comme toujours, rien de mieux que la documentation officielle pour commencer à aborder une nouvelle technologie, et pour l'installation, ça se passe par ici : <https://nodejs.org/fr/download>

Vous y trouverez les méthodes et outils d'installation pour votre système, que ce soit à partir d'un shell ou par le biais d'un installateur. A vous de choisir votre style ;-)

## NPM : Votre compagnon du quotidien

Lorsque vous installez NodeJS sur votre machine, vous installez automatiquement un utilitaire de commande associé: NPM. Ce dernier va permettre la gestion de notre projet node JS, à commencer par sa création.

Rendez-vous dans votre dossier de travail et utilisez la commande suivante afin de créer un projet NodeJS:

```
$ npm init -y
```

L'option -y vous permet de passer la phase interactive qui vous demande de préciser le nom de votre package, sa version actuelle, une description, le point d'entrée, la commande de test, le dépôt git, les mots clés, l'auteur ainsi que la licence du projet.

Cela va vous créer un fichier très important: le package.json. Il devrait avoir cette forme :

```
{
  "name": "kgcoursiut",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Voyez ce fichier comme la carte d'identité de votre projet NodeJS. Ce fichier JSON va contenir des clés précises qui détiendront chacune une information sur une partie du projet.

Par la suite, il y a deux autres sections qui seront particulièrement intéressantes et que nous détaillerons un peu plus tard : les dépendances et les scripts.

## Notre premier fichier

Nous venons de créer la carte d'identité de notre projet mais ce dernier ne fait pas grand chose. Nous allons désormais essayer d'exécuter un fichier JS qui réalisera une fonction de somme. Pour cela, créez un fichier `index.js` dans votre répertoire et ajoutez le code suivant :

```
function add(a, b) {  
  return a + b;  
}  
  
const result = add(2, 3);  
console.log(`La somme de 2 et 3 est ${result}`);
```

Pour exécuter le fichier, nous allons ensuite taper la commande suivante dans le terminal :

```
$ node index.js
```

et sans trop de surprise, vous devriez avoir un résultat relativement similaire à celui-ci :

```
$ node index.js  
La somme de 2 et 3 est 5
```

et si vous voulez un autre exemple en mode “serveur”, créez un fichier `server.js` contenant le code suivant :

```
const { createServer } = require('node:http');  
  
const hostname = '127.0.0.1';  
const port = 3000;  
  
const server = createServer((req, res) => {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello World');  
});  
  
server.listen(port, hostname, () => {  
  console.log(`Server running at http://${hostname}:${port}/`);  
});
```

En l'exécutant de la même manière, vous devriez avoir :

```
$ node server.js  
Server running at http://127.0.0.1:3000/
```

Cependant, vous remarquez que le terminal ne vous rend pas la main cette fois. Rendez-vous alors à l'adresse <http://127.0.0.1:3000> depuis votre navigateur préféré, et vous devriez avoir le classique "Hello World".

Nous reviendrons plus tard sur ces lignes de code, mais avant, poursuivons sur la structure de fonctionnement de NodeJS.

## Package et dependencies

L'un des intérêts d'utiliser NodeJS est la gestion des paquets. Au même titre que dans une distribution linux, les paquets sont des bibliothèques de différentes utilités que les développeurs vont pouvoir installer dans un projet NodeJS afin d'éviter d'avoir eux-même à réécrire des parties entières de code. A terme une application que vous développez peut elle-même être un package que vous mettrez à disposition sur le web pour d'autres utilisateurs.

Nous parlons plus tôt de NPM comme utilitaire de commande, vous avez compris qu'en réalité NPM est plutôt un gestionnaire de paquet. Par exemple, pour installer un paquet nous allons utiliser la commande suivante :

```
$ npm install nom_du_paquet
```

Dans notre cas, si nous voulons simplifier l'utilisation de fonctions mathématiques, et si nous voulons apporter un peu de couleur à notre terminal, alors nous allons donc installer deux paquets: chalk et mathjs.

```
$ npm install chalk mathjs
```

Cette commande va installer les fichiers clés du paquet dans un dossier appelé node\_modules.

Tant que nous abordons ce répertoire, une précision s'impose : Vous n'avez pas nécessairement besoin de fouiller le contenu de répertoire, dites vous juste qu'il sert à stocker vos packages externes installés. Et qu'à ce titre, si vous avez une erreur à l'exécution qui vous amène un fichier présent dans ce répertoire, il y a une TRÈS forte probabilité que l'erreur ne se situe pas vraiment là mais dans un fichier de VOTRE projet qui fait appel à ce package donc commencez par bien revoir votre trace d'exécution pour remonter à votre appel.

Si on revient à notre fichier package.json, on peut remarquer qu'il a évolué, et que la clé dependencies est désormais alimentée avec le nom de notre paquet suivi de sa version.

```
[...]  
"dependencies": {  
  "chalk": "^5.4.1",  
  "mathjs": "^14.2.0"  
}
```

La liste de tous les paquets installés par votre projet sera donc désormais indiquée dans cette clé `dependencies`.

Maintenant, nous allons utiliser ces paquets dans notre fichier `index.js`. Ajustez le pour qu'il contienne le code suivant:

```
import { add } from 'mathjs';
import chalk from 'chalk';

const a = 2;
const b = 3;

const result = add(a, b);

console.log(chalk.green(`La somme de ${a} et ${b} est ${result}`));
```

En l'exécutant de la même manière, vous devriez avoir :

```
$ node index.js
La somme de 2 et 3 est 5
```

En regardant encore un peu plus le contenu de votre projet, vous devriez à présent trouver un fichier `package-lock.json`. Ce fichier est un fichier de verrouillage de version. Il va permettre de garantir que les versions des paquets installés dans votre projet restent les mêmes. Cela permet d'éviter les problèmes de compatibilité entre les différentes versions des paquets.

Dans le cas où vous souhaitez installer les dépendances d'un projet, il est recommandé d'utiliser la commande `$ npm ci` qui va se baser sur le fichier `package-lock.json` pour installer ses dépendances.

## Distinction entre global et local

Il y a une différence importante à faire entre ces deux états :

- Un paquet installé avec `$ npm install nom_du_package` est installé localement. C'est à dire qu'il n'est accessible que dans le scope de notre projet. Et c'est uniquement une fois notre serveur node démarré que ce dernier pourra accéder aux différents packages. Il faut comprendre par là que si vous cherchez à utiliser une fonction de package dans votre terminal, alors elle ne sera pas accessible car votre terminal n'a pas accès au serveur tant qu'il n'est pas lancé.
- En revanche, un installant un paquet avec `$ npm install -g nom_du_package`, l'installation se fait de manière globale sur votre système d'exploitation directement et non pas dans un projet. Le package peut donc être utilisé dans votre terminal. Ceci peut être utile dans le cas où vous avez besoin d'un package commun à plusieurs projets par exemple.

Vous pouvez vérifier la liste des packages globaux installés grâce à la commande

```
$ npm list -g
```



## Une autre distinction : entre dependencies et devDependencies

Nous avons vu plus tôt que les dépendances étaient stockées dans la clé `dependencies` du `package.json`. Il existe une autre clé qui est `devDependencies`. Cette dernière va contenir les paquets qui ne sont pas nécessaires au fonctionnement de l'application mais qui sont utiles pour le développement. Par exemple un package de test, un package de formatage de code, ...

Pour installer un paquet et préciser qu'il doit être présent en tant que dépendance de développement, il faut utiliser l'option `-D` :

```
$ npm install -D nom_du_package
```

## Les scripts

Dans la section précédente, nous avons précisé que si nous voulions exécuter des packages installés localement au lancement de notre application, nous ne pouvions pas le faire depuis le terminal. Cependant comment faire si nous souhaitons utiliser le package `nodemon` pour le hotreload au moment où le serveur démarre ?

Pour cela, nous allons utiliser la clé `scripts` du `package.json`. Dans cette structure, il est possible d'indiquer autant de clés que vous souhaitez avoir de scripts. Vous pouvez par exemple créer un script `dev` qui correspondra à l'exécution de votre environnement de développement et un script `test` qui correspondra à l'exécution de votre environnement de test.

Quel que soit le nom que vous donnez à votre script, sa valeur sera l'équivalent de ce que vous pourriez indiquer dans votre terminal. Par exemple, l'exécution de `nodemon` dans le terminal se ferait par la commande `$ nodemon index.js`, donc pour utiliser `nodemon` dans un script il faut indiquer ceci dans le script `dev` :

```
[...]
"scripts": {
  "dev": "nodemon index.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Bien évidemment, vous aurez préalablement installé le paquet `nodemon` en tant que dépendance de développement en utilisant :

```
$ npm install -D nodemon
```

Et donc à présent, pour exécuter le script `dev` à partir du terminal, vous utiliserez désormais la commande :

```
$ npm run dev
```

Les scripts ont en réalité de nombreux autres intérêts que de simplement permettre l'exécution des packages locaux. Ils peuvent définir des variables d'environnement par exemple, réaliser une suite d'instructions avant l'exécution et permettent aussi de générer des environnements différents (`test`, `dev`, `build`, ...).

# Les fonctions fléchées (Arrow functions)

Les fonctions fléchées sont une nouvelle syntaxe pour définir des fonctions en JavaScript. Elles sont plus courtes et plus lisibles que les fonctions classiques.

Voici un exemple de fonction fléchée:

```
//Voici la version en fonction fléchée :  
const add = (a, b) => a + b;
```

```
//Ce morceau de code est équivalent à :  
function add(a, b) {  
    return a + b;  
}
```

## Récapitulatif des commandes utiles

- npm init :
  - Créer un projet nodejs et initialise son package.json
- npm init -y :
  - Variante permettant de passer la phase interactive
- npm install :
  - Installer les dépendances d'un projet
- npm ci :
  - Installer les dépendances d'un projet en se basant sur le fichier package-lock.json
- npm install nom\_du\_paquet :
  - Installer un paquet
- npm install -g nom\_du\_package :
  - Installer un paquet globalement
- npm list -g :
  - Vérifier la liste des package globaux installés
- npm run nom\_du\_script :
  - Executer un script

# ES et CommonJS

Le but de cette section est de découvrir la différence entre ECMAScript et CommonJS, leurs usages et quand les utiliser.

JavaScript est un langage qui a évolué pour devenir un standard universel dans le développement web et backend. Cependant, son écosystème a connu des étapes clés dans la manière de structurer et d'importer du code.

Deux systèmes majeurs d'organisation des modules se détachent : CommonJS et ECMAScript Modules (ESM).

## CommonJS

Avant l'introduction d'ESM, Node.js a créé son propre système de modules : CommonJS. C'est ce système qui a permis d'utiliser JavaScript sur le backend et qui est devenu la norme défacto pour Node.js.

Il a une syntaxe particulière, par exemple : Les modules sont importés avec `require` et exportés avec `module.exports` ou `exports`

```
// Importation de la fonction add depuis la librairie math.js
const math = require('./math');

// Utilisation
const result = math.add(2, 3);
console.log(`La somme est ${result}`);

// Exportation de la fonction add (depuis le fichier math.js)
module.exports = {
  add: (a, b) => a + b,
};
```

## ECMAScript

ECMAScript (ou ESM, ou ES) est une norme standardisée pour le langage JavaScript. Elle a été développée pour assurer la cohérence entre les différentes implémentations de JavaScript et étendre ses fonctionnalités de manière standard.

Avec ES6 (ou ES2015), un système de modules natif a été introduit dans JavaScript. Les modules ECMAScript permettent de charger et d'exporter des morceaux de code en utilisant les mots-clés `import` et `export`.

```
// Importation de la fonction add depuis la librairie math.js
import { add } from './math.js';

// Utilisation
const result = add(2, 3);
console.log(`La somme est ${result}`);

// Exportation de la fonction add (depuis le fichier math.js)
export function add(a, b) {
  return a + b;
}
```

## Pourquoi ces deux systèmes existent-ils ?

### ECMAScript Modules

- Origine :
  - Standardisé par ECMA pour unifier l'utilisation de modules dans tous les environnements JavaScript (navigateur et backend).
- Avantages :
  - Standard universel supporté par les navigateurs modernes et Node.js.
  - Syntaxe claire et intuitive avec import et export.
  - Optimisé pour des fonctionnalités modernes comme le tree-shaking (suppression du code inutilisé).
- Limites :
  - Exige une configuration (par exemple, ajouter "type": "module" dans package.json).
  - Chargement asynchrone des modules par défaut, ce qui peut poser des problèmes dans certains contextes.

### CommonJS

- Origine :
  - Créé spécifiquement pour Node.js afin de fournir un système de modules simple et fonctionnel pour les applications backend.
- Avantages :
  - Simple à utiliser et largement adopté dans l'écosystème Node.js.
  - Fonctionne nativement dans Node.js sans configuration supplémentaire.
  - Bonne prise en charge des modules dynamiques.
- Limites :
  - Pas adapté pour les navigateurs sans outils de transpilation.

## Différences principales entre CommonJS et ECMAScript Modules

Caractéristique	CommonJS	ECMAScript Modules (ESM)
Syntaxe d'importation	<code>const module = require(...)</code>	<code>import module from '...'</code>
Syntaxe d'exportation	<code>module.exports</code> ou <code>exports</code>	<code>export</code> ou <code>export default</code>
Chargement	Synchrone	Asynchrone
Compatibilité	Spécifique à Node.js	Universelle
Configuration supplémentaire	Aucune	Nécessite "type": "module"
Support des navigateurs	Non	Oui

# Le Typescript

Le but de cette section est de faire une légère introduction à TypeScript, une extension de JavaScript qui ajoute des types statiques pour rendre le code plus fiable, lisible et maintenable.

## Pourquoi utiliser TypeScript ?

TypeScript est une extension de JavaScript qui ajoute des types statiques. Créé par Microsoft, il est conçu pour rendre le code JavaScript plus fiable, lisible et maintenable, notamment dans les projets d'envergure.

L'un des plus grands avantages de TypeScript est la détection des erreurs lors de la compilation, contrairement à JavaScript où les erreurs ne sont souvent détectées qu'à l'exécution.

Voici un exemple simple :

```
function add(a, b) {  
  return a + b;  
}  
  
console.log(add(2, 3)); // 5  
console.log(add(2, 'three'));  
// "2three" (Erreur non détectée)
```

```
function add(a: number, b: number): number {  
  return a + b;  
}  
  
console.log(add(2, 'three'));  
// Erreur détectée à la compilation  
console.log(add(2, 3)); // 5
```

Avec TypeScript, les erreurs comme le passage d'une chaîne à la place d'un nombre sont évitées à l'avance.

## Installation

Pour installer Typescript, comme tous packages, deux solutions s'offrent à vous. Soit vous l'installez globalement sur votre machine, soit vous l'ajoutez en tant que dépendance de votre projet.

```
# Installation globale  
$ npm install -g typescript
```

```
# Installation locale  
$ npm install -d typescript
```

Vous verrez qu'un certain nombre d'autres paquets sont également installés en même temps que typescript.

## Configuration

Pour configurer TypeScript, vous pouvez créer une configuration par défaut en utilisant la commande suivante :

```
# Dans Le cadre globale  
$ tsc --init
```

```
# Dans Le cadre Local  
$ npx tsc --init
```

Cette commande va générer un fichier `tsconfig.json` qui contient la configuration de TypeScript pour votre projet. Vous pouvez personnaliser ce fichier en fonction de vos besoins. Nous n'allons pas rentrer dans le détail de la configuration de Typescript ici, mais vous pouvez consulter la documentation officielle.

## Compilation d'un fichier TypeScript

TypeScript utilise l'extension `.ts` pour les fichiers. Pour commencer, il nous faut donc modifier notre fichier `index.js` en `index.ts` pour le transformer en fichier TypeScript.

Cependant, pour pouvoir lancer notre application NodeJS, nous devons compiler notre fichier TypeScript en JavaScript. Pour cela, nous utilisons la commande `tsc` (TypeScript Compiler).

```
$ tsc index.ts
```

Ainsi, si vous exécutez sur le code incorrect que nous avons vu plus haut, vous devriez obtenir une erreur à la compilation

```
PS C:\Users\guerr\OneDrive\Bureau\NodeJS\kgcoursiut> tsc index.ts
index.ts:5:22 - error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.

5   console.log(add(2, 'three')); // Erreur détectée à la compilation
                        ~~~~~

Found 1 error in index.ts:5
```

Alors qu'avec le code correct, la compilation se déroule sans problème et génère le code suivant dans le fichier `index.js` :

```
function add(a, b) {
  return a + b;
}
// console.log(add(2, 'three')); // Erreur détectée à la compilation
console.log(add(2, 3)); // 5
```

Pour vous simplifier la commande, vous pouvez ajouter un script nommé "build" dans le fichier `package.json` et qui exécutera cette compilation.

```
[...]
"scripts": {
  "dev": "nodemon index.js",
  "build": "tsc index.ts",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Ainsi, la commande `$ npm run build` va générer un fichier `index.js` à partir de `index.ts` que vous pourrez ensuite exécuter avec NodeJS comme nous l'avons fait précédemment.

## Gestion du hot-reload

Pour éviter de devoir compiler à chaque modification de fichier, vous pouvez utiliser un outil comme ts-node qui permet d'exécuter directement des fichiers TypeScript sans les compiler.

```
$ npm install -d ts-node
```

Vous pouvez ensuite exécuter votre fichier TypeScript directement avec ts-node.

```
$ ts-node index.ts
```

Cependant ts-node ne fait qu'exécuter le code sans le compiler. Si le fichier est modifié, il faudra relancer la commande. Nous allons donc le coupler avec nodemon que nous avons vu précédemment.

```
[...]
"scripts": {
  "dev": "nodemon --exec ts-node ./index.ts",
  "build": "tsc index.ts",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

## Notions de base de TypeScript

Typescript ne permet pas simplement de déclarer des types ou compiler du code. Il ajoute également des fonctionnalités qui rendent le code plus lisible et maintenable.

Certaines de ces notions doivent vous être familières si vous avez déjà travaillé avec des langages de programmation comme Java ou C#.

### Typage des variables

TypeScript permet de définir explicitement les types de variables.

```
let p_name: string = 'Alice';
let p_age: number = 25;
let isStudent: boolean = true;
```

### Typage des fonctions

Ajoutez des types pour les arguments et la valeur de retour.

```
function greet(name: string): string {
  return `Hello, ${name}`;
}
```



## Les Interfaces

Les interfaces permettent de définir des structures pour vos objets.

```
interface User {  
  name: string;  
  age: number;  
  isAdmin?: boolean; // Propriété optionnelle  
}  
  
const user: User = { name: 'Bob', age: 30 };
```

## Les Classes

TypeScript ajoute des notions de programmation orientée objet comme les classes.

```
class Person {  
  name: string;  
  age: number;  
  
  constructor(name: string, age: number) {  
    this.name = name;  
    this.age = age;  
  }  
  
  greet(): string {  
    return `Hi, I'm ${this.name}`;  
  }  
}  
  
const person = new Person('Alice', 25);  
console.log(person.greet());
```

## Les Types avancés / Union Types

Un argument peut accepter plusieurs types.

```
function printId(id: number | string): void {  
  console.log(`ID: ${id}`);  
}
```

## Les Type Aliases

Créez des alias pour des types complexes.

```
type Point = { x: number; y: number };  
  
const point: Point = { x: 10, y: 20 };
```

## Enums

Les énumérations permettent de définir un ensemble de constantes nommées.

```
enum Direction {  
  Up = 'UP',  
  Down = 'DOWN',  
  Left = 'LEFT',  
  Right = 'RIGHT',  
}
```

## Les packages @types

Certains packages JavaScript n'ont pas de type TypeScript par défaut. Pour les utiliser dans un projet TypeScript, vous pouvez installer des packages @types. Ce sont des packages que la communauté TypeScript a créés pour ajouter des types à des packages JavaScript existants.

```
$ npm install -d @types/nom_du_paquet
```

Par exemple pour récupérer les types de node:

```
$ npm install -d @types/node
```

Dans les prochains chapitres nous prendrons l'habitude d'installer aussi les types des packages que nous utilisons.

## Aller plus loin

TypeScript est un langage puissant qui offre de nombreuses fonctionnalités pour améliorer la qualité et la maintenabilité de votre code. Le comportement de compilation de TypeScript peut par exemple être personnalisé. Nous verrons plus en profondeur ces notions dans le module dédié à VueJS.

# Les requêtes HTTP

Le but de cette section est de comprendre les requêtes HTTP et comment les gérer en NodeJS.

A chaque fois que vous tentez d'accéder à une page web vous réalisez une requête HTTP. Cette requête va contacter l'hébergeur de la page que vous ciblez et va attendre une réponse qui permettra l'affichage du site. Il est important de comprendre que lorsque vous cherchez à accéder à un site, pour chaque fichier requis vous réaliserez une requête HTTP pour obtenir l'information.

Les gros sites pouvant rapidement atteindre des masses critiques de requêtes, il est important pour eux d'optimiser les appels HTTP au maximum afin de gagner en performance.

Dans le cas d'une API le nombre d'appels HTTP est censé être minime, voire unique.

## Structure d'une requête HTTP

Maintenant que nous comprenons un peu mieux à quoi sert une requête HTTP, il est important de voir comment elle est structurée. Il ne s'agit pas ici d'un cours approfondi sur HTTP, aussi nous passerons en revue uniquement les éléments les plus importants de ce protocole.

### Types de requête

Lorsque vous réalisez une requête HTTP elle possède un type. Par défaut il s'agit d'une requête GET. Elle permet d'obtenir des informations en les ramenant au client. Mais il existe de nombreux autres types, dont voici une liste des plus fréquemment rencontrés.

- POST :
  - Il s'agit de la requête par défaut lorsque l'on veut envoyer des données au serveur, lors d'une création ou la soumission d'un formulaire par exemple.
- DELETE :
  - La requête utilisée pour indiquer la suppression / destruction de données.
- PATCH :
  - C'est une requête utile en cas de modification de données.
- HEAD :
  - Elle permet de ne récupérer que les informations d'en-tête et donc de limiter la taille de la réponse. Utile lorsque l'on souhaite faire des vérifications avant une requête GET par exemple pour connaître la taille de la page en amont.

## Code HTTP

Une fois la réponse reçue elle est souvent accompagnée d'un code qui permet de rapidement savoir la nature de la réponse. Vous connaissez sûrement l'un des plus connus: le code 404 Not Found.

Les codes HTTP se décomposent d'abord avec leur chiffre de centaine qui correspond chacun à un type. Par exemple le 4 de 400 ici correspond à une erreur client. Et ensuite on incrémente un nombre pour avoir du détail, ici 04 pour dire ressource non trouvée (not found).

Code	Signification	Description
200	OK	La requête a été traitée avec succès. Cela signifie généralement que la requête a réussi et que la réponse contient les données demandées.
201	Created	La requête a été traitée avec succès, et une nouvelle ressource a été créée. Ceci est couramment utilisé pour les requêtes POST.
204	No Content	La requête a réussi, mais il n'y a pas de contenu à renvoyer. Par exemple, après une suppression (DELETE), la réponse peut être vide.
400	Bad Request	La requête est mal formulée ou manque d'informations nécessaires. Ce code indique généralement une erreur du côté du client.
401	Unauthorized	La requête nécessite une authentification. Ce code est souvent renvoyé lorsqu'une API requiert une clé d'API ou un jeton d'authentification.
403	Forbidden	Le serveur comprend la requête, mais il refuse de l'exécuter. Cela peut être dû à des permissions insuffisantes.
404	Not Found	La ressource demandée n'a pas été trouvée sur le serveur. Cela est couramment utilisé pour indiquer que l'URL ou l'endpoint n'existe pas.
405	Method Not Allowed	La méthode HTTP utilisée (par exemple, GET, POST, PUT, DELETE) n'est pas autorisée pour la ressource spécifiée.
500	Internal Server Error	Il y a une erreur côté serveur. Cela signifie que quelque chose s'est mal passé, mais le serveur ne peut pas donner plus de détails.

Vous trouverez le détail des codes HTTP sur <https://developer.mozilla.org/fr/docs/Web/HTTP/Status>

# Réaliser une requête HTTP en NodeJS

Le but d'un serveur NodeJS est de pouvoir répondre à des requêtes HTTP. Et pour cela, nous allons utiliser le package natif de NodeJS `'http'`. Ce package nous permet de créer un serveur HTTP et de répondre à des requêtes. Nous avons déjà vu comment créer le serveur et l'écouter. Maintenant nous allons l'adapter pour qu'il soit capable de créer des réponses en fonction du type de requête.

Nous allons donc récupérer l'objet `req` qui est passé en paramètre de notre callback de création de serveur. Cet objet contient de nombreuses informations sur la requête, dont la méthode utilisée (`method`). Nous allons donc vérifier la méthode de la requête et renvoyer une réponse en fonction via l'objet `res`.

```
import http from 'http';

// Création du serveur HTTP
const server = http.createServer((req, res) => {
  // Vérification de la méthode de la requête
  if (req.method === 'GET') {
    // Définition du code de statut de la réponse à 200 (OK)
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    // Envoi de la réponse
    res.end('Requête GET réussie avec code 200\n');
  } else {
    // Si la méthode de la requête n'est pas GET, renvoyer un code 405 (Method Not
    Allowed)
    res.writeHead(405, { 'Content-Type': 'text/plain' });
    res.end('Méthode non autorisée\n');
  }
});

// Démarrage du serveur sur le port 3000
server.listen(3000, () => {
  console.log("Serveur en cours d'écoute sur le port 3000...");
});
```

Pour rappel, une fonction callback est une fonction qui est passée en argument à une autre fonction et qui est ensuite exécutée à un moment donné, généralement après que cette fonction ait terminé son traitement.

En d'autres termes, c'est une fonction qui "attend" d'être appelée une fois qu'une tâche asynchrone (comme une requête HTTP ou une opération de lecture de fichier) est terminée.

# Express

Le but de cette section est de découvrir le framework Express pour NodeJS.

## Initialisation

Express est un framework qui permet de faciliter la création de serveur HTTP. En effet, même s'il est possible de créer notre propre serveur web de A à Z en définissant chaque méthode HTTP, chaque route possible, cependant Express vous permettra de réaliser ce travail tellement plus facilement que vous l'utiliserez presque systématiquement lors de la réalisation d'un backend en NodeJS.

Comme pour les autres outils, il faut commencer par installer express avec la commande:

```
$ npm install express @types/express
```

Ensuite dans notre fichier main (souvent index.js) nous allons ajouter les lignes suivantes:

```
import express from 'express';

export const app = express();

const port = 3000;

app.listen(port, () => {
  console.log(`Mon serveur démarre sur le port ${port}`);
});
```

Cette partie de code est pour le moment très similaire à ce que nous faisons précédemment avec le package `'http'`, mais patience c'est après qu'express va révéler tout son intérêt.

Et l'app express est exportée afin de pouvoir l'utiliser dans d'autres fichiers. Cela se révèlera utile lorsque nous voudrons tester notre application plus tard.

## Routes

Pour le moment, Express nous a permis de créer un serveur, ce que le package HTTP pouvait déjà faire. Mais pour créer réellement une API il va falloir gérer des cas de lecture, écriture, modification, suppression, ...

Afin de gérer ces cas nous allons passer par des requêtes HTTP, dont chacune correspond à un de nos besoins. Ainsi, pour accéder à une donnée, une requête GET sera réalisée sur l'API (c'est-à-dire sur le serveur NodeJS). De même, pour supprimer une donnée en particulier cela passera par une requête DELETE en précisant la donnée qui doit être supprimée.

Et c'est là que le framework Express prend toute sa valeur, car il possède déjà une liste de fonctions permettant de gérer ces cas.

Sur les documentations vous trouverez souvent les routes définies de cette manière: `GET:/non-de-La-route`. Cela signifie que pour accéder à cette route il faudra faire une requête `GET` sur `/nom-de-La-route`.

## Lecture via GET

Admettons que vous travaillez sur une API permettant de gérer une liste d'élèves. L'application front a besoin d'accéder à cette liste d'élèves et va donc demander une requête `GET` à votre API (le serveur hébergé sur `localhost:3000`) et ce dernier devra lui rendre la fameuse liste.

Mais il va également parfois être nécessaire d'obtenir seulement un seul élève. Et dans les deux cas il s'agit d'une requête `GET`. Il faut donc créer des routes permettant de différencier ces cas, ce qui correspond en fait à des url supplémentaires.

Pour définir une route `GET` nous allons utiliser la méthode du même nom d'express.

```
import express from 'express';
import { Request, Response } from 'express';

export const app = express();

const port = 3000;

app.listen(port, () => {
  console.log(`Mon serveur démarre sur le port ${port}`);
});

// Route pour obtenir la liste des utilisateurs
app.get('/users', (_req: Request, res: Response) => {
  res.status(200).send('Liste des utilisateurs');
});

// Route pour obtenir un utilisateur en particulier
app.get('/user/:id', (_req: Request, res: Response) => {
  res.status(200).send(`Utilisateur ${_req.params.id}`);
});
```

Vous aurez remarqué que cette syntaxe est typée donc nous n'utilisons plus directement le fichier js mais nous passons par le fichier `index.ts`, et que nous pouvons exécuter en utilisant notre script `dev`.

```
$ npm run dev
```

## Création via POST

Pour créer une ressource nous allons utiliser la méthode POST. Cette méthode est utilisée pour envoyer des données au serveur. Par exemple, dans le cas d'un formulaire d'inscription, les données du formulaire seront envoyées au serveur avec une requête POST.

```
[...]
app.post('/user', (req: Request, res: Response) => {
  res.send(`Ajout de l'utilisateur ${req.body.name}`);
});
```

## Middleware

Les middlewares sont des morceaux de code qui seront exécutés lors de chaque requête. Les middlewares peuvent se révéler utiles lorsque vous avez des opérations communes à réaliser avant ou après chaque requête. Par exemple, si vous souhaitez mettre en place un système d'horodatage des requêtes.

```
// import { Request, Response } from 'express';
import { Request, Response, NextFunction } from 'express';
[...]
app.use((req: Request, _res: Response, next: NextFunction) => {
  const timestamp = new Date().toISOString();
  console.log(`[${timestamp}] Requête reçue : ${req.method} ${req.url}`);
  next(); // Passe à la prochaine fonction middleware ou route
});
[...]
```

## Body Parsing

Il est cependant possible que vous ayez des difficultés lorsque la requête va essayer de lire le body. Dans ce cas, il faut utiliser un parser qui va formater la chaîne de caractère reçue pour qu'elle soit interprétable par Express.

Vous pouvez consulter les informations sur cet outil en consultant la ressource officielle de [expressjs.com](https://expressjs.com/en/resources/middleware/body-parser.html) à propos du body-parser

On va ensuite simplement ajouter un middleware à Express qui demandera d'utiliser ce parser. Ainsi toutes les requêtes qui arriveront sur mon serveur seront parsées.



Par exemple, pour parser un body qui est au format JSON.

```
import express from 'express';
import { Request, Response } from 'express';

export const app = express();
app.use(express.json());

app.post('/user', (req: Request, res: Response) => {
  res.send(`Ajout de l'utilisateur ${req.body.name}`);
});
```

Pour simuler vos requêtes, vous pouvez utiliser l'outil Postman <https://www.postman.com>

# ORM et Database

Le but de cette section est de faire une introduction aux ORM (Object-Relational Mapping) dans Node.js, avec un focus sur Prisma et l'importance d'utiliser un ORM pour la gestion des bases de données

Pour le moment, les requêtes HTTP que nous avons mises en place ne permettent pas la persistance de données. Cela signifie que dans le cas d'un redémarrage du serveur, toutes les données sont perdues. Pour éviter cela, nous allons utiliser un ORM pour sa simplicité d'utilisation.

## Pourquoi utiliser un ORM ?

Les ORM (Object-Relational Mapping) sont des outils permettant de gérer les interactions entre une base de données relationnelle et le code d'une application. Ils simplifient les requêtes et la gestion des données en traduisant des objets dans votre code en enregistrements dans la base de données, et vice-versa.

Dans un environnement Node.js, les ORM jouent donc un rôle crucial en permettant de :

- Simplifier les requêtes SQL complexes
- Gérer les migrations de manière fluide
- Éviter l'écriture de SQL brut, ce qui améliore la maintenabilité du code
- Accéder aux bases de données de manière déclarative et sécurisée

L'utilisation d'un ORM présente donc plusieurs avantages importants, notamment :

- Une sécurité renforcée : L'ORM permet de réduire les risques d'injection SQL en générant automatiquement des requêtes sécurisées.
- L'abstraction des bases de données : Grâce à l'ORM, vous pouvez interagir avec la base de données sans avoir besoin de connaître en détail le langage SQL, rendant le code plus lisible et maintenable.
- La gestion des migrations : L'ORM facilite la gestion des évolutions de la base de données, ce qui est crucial dans des environnements agiles où le modèle de données change fréquemment.
- Des performances optimisées : Les ORM modernes, comme Prisma, sont conçus pour générer des requêtes SQL performantes, souvent plus rapides que celles écrites manuellement par des développeurs.

Et si vous préférez aborder ce sujet à partir des raisons d'éviter de faire sans un ORM :

- Code difficile à maintenir : Le SQL peut rapidement devenir difficile à comprendre et à maintenir, surtout dans les projets de grande envergure avec plusieurs développeurs.
- Risque de sécurité : L'absence de mécanismes de protection contre les injections SQL rend les applications vulnérables.
- Migrations manuelles : La gestion manuelle des migrations de base de données devient rapidement complexe à mesure que le projet grandit.
- Portabilité limitée : Si vous devez changer de SGBD (par exemple passer de MySQL à PostgreSQL), les requêtes SQL devront être réécrites et adaptées à la nouvelle base.

En conclusion, l'utilisation d'un ORM comme Prisma permet de surmonter ces problèmes et de gagner du temps tout en assurant la sécurité et la maintenabilité du code.

# Prisma

Le but de cette section est de faire une introduction à Prisma, ses concepts de base, et ses commandes essentielles dans un projet Node.js

Prisma est un ORM (Object-Relational Mapping) moderne qui simplifie la gestion des bases de données dans les projets Node.js. Contrairement aux ORM traditionnels, Prisma adopte une approche déclarative qui facilite la synchronisation entre votre code et votre base de données.

## Pourquoi utiliser Prisma ?

Prisma offre de nombreux avantages :

- Facilité d'utilisation : Écriture simple et intuitive des requêtes grâce au Prisma Client.
- Type safety : Prisma génère automatiquement des types TypeScript basés sur votre schéma, réduisant les erreurs.
- Automatisation des migrations : Gestion des changements dans votre base de données avec un minimum d'efforts.
- Performances optimisées : Prisma utilise des requêtes SQL optimisées.

## Concepts de base de Prisma

Prisma repose sur trois composants principaux :

- `prisma/schema.prisma` : Un fichier `.prisma` qui définit le modèle de données, les relations, et les configurations de la base de données.
- Prisma Client : Un client généré automatiquement pour interagir avec la base de données à travers votre code.
- Prisma CLI : Un outil en ligne de commande pour gérer le schéma, les migrations, et bien plus.

## Installation et initialisation

Pour ajouter Prisma à votre projet, vous pouvez utiliser :

```
$ npm install prisma --save-dev
$ npm install @prisma/client
```

Ensuite, initialisez Prisma dans votre projet :

```
$ npx prisma init
```

Cette commande crée deux fichiers principaux :

- `prisma/schema.prisma` : Le fichier principal pour définir votre modèle.
- `.env` : Un fichier de configuration pour les variables d'environnement, comme l'URL de votre base de données.

# Comprendre les essentiels

## Connecteur de base de données

Prisma prend en charge plusieurs connecteurs de base de données, tels que MySQL, PostgreSQL, SQLite, et SQL Server. Vous pouvez spécifier le connecteur dans le fichier `schema.prisma` et le configurer dans le fichier `.env`:

<pre>// This is your Prisma schema file, // learn more about it in the docs: https://pris.ly/d/prisma-schema  // Looking for ways to speed up your queries, or scale easily with your serverless or edge functions? // Try Prisma Accelerate: https://pris.ly/cli/accelerate-init  generator client {   provider = "prisma-client-js" }  datasource db {   provider = "sqlite"   url      = env("DATABASE_URL") }</pre>	<pre># Environment variables declared in this file are automatically made available to Prisma. # See the documentation for more detail: https://pris.ly/d/prisma-schema#accessing- environment-variables-from-the-schema  # Prisma supports the native connection string format for PostgreSQL, MySQL, SQLite, SQL Server, MongoDB and CockroachDB. # See the documentation for all the connection string options: https://pris.ly/d/connection-strings  DATABASE_URL="file:./dev.db"</pre>
---	---

## Migration

Une migration Prisma correspond à un ensemble de modifications appliquées à une base de données pour aligner sa structure (tables, colonnes, relations, etc.) avec le modèle de données défini dans le fichier `schema.prisma`.

Ce processus est utilisé pour synchroniser le modèle Prisma avec la base de données, particulièrement dans les projets où le modèle évolue fréquemment.

Créer un schéma :

Dans le fichier `schema.prisma` vous pouvez définir un modèle correspondant à une table.

```
[...]
model User {
  id    Int    @id @default(autoincrement())
  email String @unique
  name  String
  password String
  posts Post[] @relation("Post_user") // Relation avec le modèle Post
}

model Post {
  id      Int @id @default(autoincrement())
  title   String
  content String?
  author  User @relation(name: "Post_user", fields: [authorId], references: [id])
  authorId Int
}
```

Si vous voulez plus de détails sur la gestion des relations, vous pouvez consulter la documentation officielle : <https://www.prisma.io/docs>

Créer une migration :

```
$ npx prisma migrate dev --name init
```

Cette commande :

- Génère un fichier de migration contenant les instructions SQL nécessaires.
- Applique les modifications à votre base de données.

## Seed

Le seeding consiste à insérer des données initiales dans votre base, par exemple pour des tests ou un environnement de développement.

Pour la configuration du script de seed, dans `package.json`, il est nécessaire d'ajouter :

```
[...] ,
"prisma": {
  "seed": "ts-node prisma/seed.ts"
}
}
```

Créer le script de seed : Exemple de fichier prisma/seed.ts :

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

async function main() {
  // Suppression de tous les posts
  await prisma.user.deleteMany();
  await prisma.post.deleteMany();

  // Réinitialisation de l'auto-incrémentation sur SQLite
  await prisma.$executeRaw`DELETE FROM sqlite_sequence WHERE name='User'`;
  await prisma.$executeRaw`DELETE FROM sqlite_sequence WHERE name='Post'`;

  // Création de plusieurs utilisateurs avec createMany
  await prisma.user.createMany({
    data: [
      { name: 'John Doe', }, { name: 'Jane Smith', },
    ],
  });

  // Création de posts pour un utilisateur spécifique avec create
  await prisma.post.create({
    data: {
      title: 'Premier Post de John',
      // author: { connect: { email: 'j.doe@iutmail.com' }, }
      authorId: 1, // ID de l'utilisateur John Doe
    },
  });

  // Création de plusieurs posts avec createMany
  await prisma.post.createMany({
    data: [
      {
        title: 'Premier Post de John',
        // sets userId of Profile record
        // author: { connect: { email: 'j.doe@iutmail.com' }, }
        authorId: 1, // ID de l'utilisateur John Doe
      },
      {
        title: 'Deuxième post de Jane',

```

```

        // author: { connect: { email: 'j.doe@iutmail.com' }, } // sets userId of
Profile record
        authorId: 2,
    },

],
});
}

main()
    .catch((e) => {
        throw e;
    })
    .finally(async () => {
        await prisma.$disconnect();
    });

```

Exécuter le seed :

```
$ npx prisma db seed
```

## Generate

La commande generate sert à recréer le Prisma Client lorsque votre schéma change. Sans cette commande, les modifications dans schema.prisma ne seront pas reflétées dans votre code.

Cela se fait par la commande :

```
$ npx prisma generate
```

Il est nécessaire de l'utiliser après toute modification du fichier schema.prisma ou lors de l'installation de nouvelles dépendances Prisma.

## Utiliser le client

Dans votre fichier Typescript, vous pouvez utiliser le Prisma Client pour interagir avec la base de données.

```

[...]
```

```

import { PrismaClient } from '@prisma/client';
const prisma = new PrismaClient();
[...]
```

```
app.get('/users', async (_req, res) => {
  // Récupère tous les utilisateurs et leurs posts
  const users = await prisma.user.findMany({ include: { posts: true } });
  res.status(200).send(users);
});
```

La propriété `include` permet de spécifier les relations à inclure dans la requête.

## Pattern singleton

Pour éviter de créer une nouvelle instance du Prisma Client à chaque requête, il est recommandé d'utiliser un pattern singleton pour garantir une seule instance partagée entre les requêtes. Faisons cela dans un fichier `./client.ts` :

```
import { PrismaClient } from '@prisma/client';

class DBClient {
  public prisma: PrismaClient
  private static instance: DBClient
  private constructor() {
    this.prisma = new PrismaClient()
  }

  public static getInstance = () => {
    if (!DBClient.instance) {
      DBClient.instance = new DBClient()
    }
    return DBClient.instance
  }
}

export default DBClient
```

et son utilisation se fera par :

```
[...]
import DBClient from './client'
const prisma = DBClient.getInstance().prisma
[...]
app.get('/users', async (_req, res) => {
  const users = await prisma.user.findMany({ include: { posts: true } }); // Récupère
  tous les utilisateurs et leurs posts
  res.status(200).send(users);
});
```



# Prisma studio

Prisma Studio est un outil de visualisation de données qui vous permet d'explorer et de modifier les données de votre base de données. Pour lancer Prisma Studio, exécutez :

```
$ npx prisma studio
```

## Commandes Utiles de prisma client

### Prisma Client

- `findMany` : Récupérer plusieurs enregistrements.
- `findUnique` : Récupérer un enregistrement unique.
- `create` : Créer un nouvel enregistrement.
- `update` : Mettre à jour un enregistrement existant.
- `delete` : Supprimer un enregistrement.

### Prisma CLI

- `prisma migrate` : Gérer les migrations.
- `prisma db seed` : Exécuter le script de seed.
- `prisma generate` : Régénérer le Prisma Client.
- `prisma studio` : Lancer Prisma Studio.

# Architecture projet

## Structurer son projet NodeJS

Maintenant que les bases de Node et de la gestion d'API sont acquises il va être temps de structurer un peu mieux notre projet.

### Routes

Les routes seront les points d'entrée de notre API. Elles seront définies dans un fichier `*.router.js` et seront appelées dans l'index. Elles pourront appeler des middlewares et des controllers pour répondre aux requêtes HTTP.

### Middlewares

Les middlewares seront des fonctions qui seront appelées avant les routes. Elles pourront vérifier des informations, modifier des données ou encore appeler des fonctions externes. Elles seront appelées dans les routes via la méthode `use` de `express`. Elles seront définies dans des fichiers `*.middleware.js`.

### Controllers

Les contrôleurs seront les fonctions qui contiendront la logique métier de notre application. Ils seront appelés par les routes et pourront appeler les modèles pour interagir avec la base de données. Ils seront définis dans des fichiers `*.controller.js`.

### Models

Les modèles seront les fonctions qui permettent d'interagir avec la base de données. Ils seront appelés par les controllers et pourront faire des requêtes à la base de données. Ils seront définis dans un fichier `*.model.js`.

Cependant, comme nous utilisons `prisma` qui sert de modèle nous n'aurons pas besoin de créer de modèle pour l'instant.

Ce qui donne une structure telle que :

```
project-root/
├── src
│   ├── common
│   │   └── *.middleware.js
│   ├── *
│   ├── *.middleware.js
│   ├── *.model.js (si nécessaire)
│   ├── *.controller.js
│   ├── *.router.js
│   └── index.js
├── package.json
├── .env
├── prisma
├── node_modules
└── tsconfig.json
```

Chaque partie de notre projet sera désormais proprement découpée.

L'index fera appel aux routes en tant que middleware, qui feront appel aux fonctions définies dans au niveau des controllers, puis ces controllers appelleront les modèles qui interagiront avec la base de données.

Si on veut illustrer la partie user, les appels peuvent se retrouver de la manière suivante :

Dans le fichier index.ts

```
[...]
import { userRouter } from './user/user.router';
[...]
app.use('/users', userRouter);
```

Dans le fichier user/user.router.ts

```
import { Router } from 'express';
import { getUsers } from './user.controller';

export const userRouter = Router();

// Route pour obtenir la liste des utilisateurs
userRouter.get('/', getUsers);
```

Dans le fichier user/user.controller.ts

```
import { Request, Response } from 'express';
[...]

export const getUsers = async (_req: Request, res: Response) => {
  res.status(200).send('Liste des utilisateurs');
}
```

# Exercice de prise en main :

Structurez votre projet en respectant les éléments ci-dessus.

Créez les éléments nécessaires permettant :

- de lister les users,
- de consulter un utilisateur spécifique
- de modifier un utilisateur spécifique
- de supprimer un utilisateur spécifique
- de lister les posts,
- de lister les posts d'un utilisateur spécifique,
- de consulter un post spécifique
- de modifier un post spécifique
- de supprimer un post spécifique

Vous pourrez tester vos requêtes en utilisant l'outil postman

## Extra Part : Rappel GIT

Les principales commandes git :

- git status :
  - Liste les fichiers modifiés localement
- git add fichier1 [fichier2] [...] :
  - Marque les fichiers précisés comme faisant partie du commit à venir
- git commit -m 'titre\_de\_mon\_commit' :
  - Valide les modifications sur les fichiers marqués et les regroupe dans un commit ayant pour titre le message défini
- git push :
  - Transmet au serveur distant le commit
- git pull :
  - Met à jour le répertoire local à partir du serveur distant

Lorsque vous utilisez GIT, l'idée est de valider par un commit chaque étape permettant de passer d'une version stable à une autre version stable. Par exemple, lorsque vous mettez en place la structure pour une nouvelle entité avec un nouveau repertoire au nom de cette entité, contenant les fichiers router.js et controller.js, ainsi que l'ajout du middleware

# JsonWebToken

Le but de cette section est de faire une introduction à l'utilisation des tokens d'authentification avec NodeJS.

## Structure

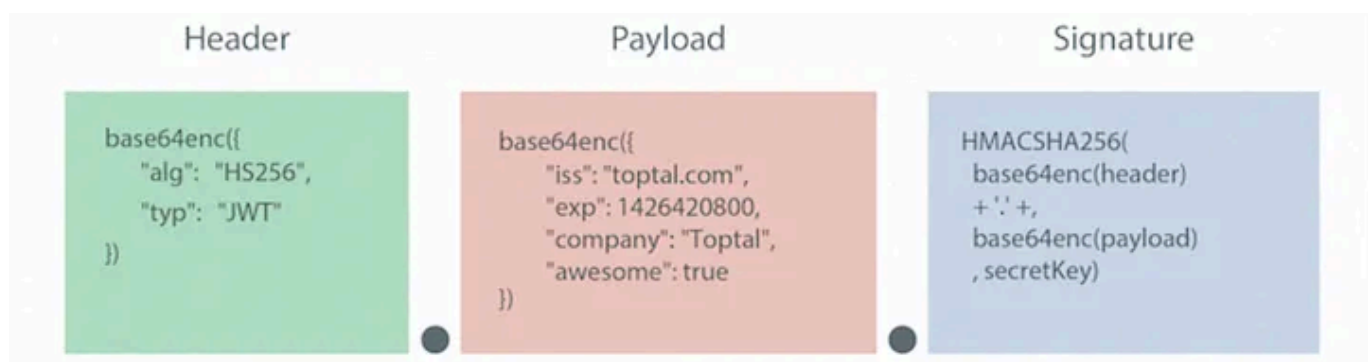
Les tokens d'authentification sont des jetons qui sont obtenus lors de la connexion d'un utilisateur. Ce dernier reçoit une clé cryptée qui contient des informations permettant de l'identifier. Ces jetons sont souvent stockés dans le LocalStorage ou les Cookies sur les navigateurs.

À chaque fois que l'utilisateur tente de se rendre sur des ressources sécurisées, le serveur vérifie la validité du token. Si le token est correct et correspond bien à l'utilisateur alors le serveur autorise l'accès à la ressource sinon il le rejettera (on se retrouve dans un cas d'erreur 401 Unauthorized).

Les tokens sont donc très pratiques car ils évitent à un utilisateur de devoir se reconnecter systématiquement à chaque requête au serveur. D'autant qu'il est possible de leur donner une durée de vie, permettant ainsi d'éviter la reconnexion pendant ce laps de temps.

Pour utiliser les tokens avec nodeJS, nous utilisons les jsonwebtoken ou JWT. Il s'agit d'un format standard et vous trouverez toutes les informations utiles et nécessaires sur <https://jwt.io/>. Sa structure se décompose en 3 parties:

- **HEADER:** c'est l'en-tête qui définit le type de token ainsi que son algorithme d'encryptage de signature. C'est un objet JSON.
- **PAYLOAD:** c'est l'élément qui possède les données (data) que l'on souhaite stocker dans le JWT, comme l'id utilisateur, son rôle, (...). C'est un objet JSON.
- **SIGNATURE:** C'est une signature numérique qui permet le chiffrement et le déchiffrement de notre JWT. On l'obtient en chiffrant le HEADER et le PAYLOAD avec l'encodage base64url. Ensuite, on les concatène en les séparant par un point. On obtient la signature de ce résultat avec l'algorithme choisi. Cette signature est ajoutée au résultat de la même manière (encodée et séparée par un point). Généralement, on rajoute à cela une clé de chiffrement définie par nos soins. Le chiffrement est capital car il permet de vérifier l'intégrité du token.



Si vous définissez une clé de chiffrement dans votre projet veillez à bien la définir dans vos variables d'environnement pour que personne ne puisse y accéder depuis le cloud, cela serait catastrophique car on pourrait pirater votre application.

## Créer un JWT

Pour utiliser ce standard nous utiliserons le package `jsonwebtoken`. Le `jsonwebtoken` sera ensuite appelé dans l'endpoint de connexion (généralement l'endpoint `/auth`) grâce à sa fonction `sign` qui sert à chiffrer un nouveau token.

```
$ npm install jsonwebtoken
$ npm install -D @types/jsonwebtoken
```

Nous allons aborder l'exemple de la fonction qui réalise plusieurs opérations et finit par renvoyer l'utilisateur renseigné dans la requête. En plus des données de l'utilisateur, nous allons chercher à renvoyer dans la réponse un JWT.

src/user/user.controller.ts

```
export const login = async (req: Request, res: Response) => {
  const { username, password } = req.body;

  if (username !== 'admin' || password !== 'password') {
    res.status(401).send('Identifiants invalides');
    return;
  }
  const token = jwt.sign(
    { username: username }, // Payload
    process.env.JWT_SECRET as jwt.Secret, // Secret
    { expiresIn: process.env.JWT_EXPIRES_IN } // Expiration"
  );

  res.status(200).json({
    token,
  });
};
```

Les variables d'environnement `JWT_SECRET` et `JWT_EXPIRES_IN` doivent être définies dans un fichier `.env` à la racine de votre projet.

## Vérifier un JWT

Si tout va bien, la réponse à la demande de connexion doit renvoyer un JWT. Votre application (ou vous-même) allez désormais stocker ce JWT et l'utiliser dans vos prochaines requêtes afin de s'assurer que les endpoints nécessitant une connexion aient connaissance de votre token et vous délivrent l'accès à leurs fonctions.

Afin qu'un endpoint limite ses accès uniquement à un utilisateur possédant un JWT valide, nous allons créer un middleware d'authentification qui s'exécutera avant les contrôleurs.

Pour commencer, il est de coutume d'envoyer le jwt dans l'entête de votre requête et plus spécifiquement dans la clé `authorization`. On va donc récupérer le token contenu dans cette clé puis le vérifier à l'aide de la fonction `verify` qui prendra en paramètre le JWT et la clé de chiffrement secrète `SECRET_TOKEN`.

`src/common/jwt.middleware.ts`

```
import { NextFunction, Response, Request } from 'express';
import jwt from 'jsonwebtoken';

export const verifyJWT = (req: Request, res: Response, next: NextFunction) => {
  const authHeader = req.headers.authorization;
  if (authHeader) {
    const token = authHeader.split(' ')[1];
    const decodedToken = jwt.verify(
      token,
      process.env.JWT_SECRET as jwt.Secret
    ) as {
      userId: string;
    };
    const userId = decodedToken.userId;
    req.query = {
      userId: userId,
    };
    next();
  } else {
    res.sendStatus(401);
  }
};
```

## Point sécurité

- Ne pas stocker de JWT sensibles dans `localStorage` : Préférez les cookies sécurisés et `HTTP-only`.
- Configurer une durée de vie appropriée : Les tokens doivent expirer rapidement (`exp`).
- Utiliser un algorithme de signature robuste : Préférer `RS256` ou `ES256` (asymétrique) à `HS256`.
- Vérifier toujours la signature et les claims : Assurez-vous que le token est valide et les informations dans le payload sont autorisées.

## Jest

Le but de cette section est de faire une introduction à la réalisation des tests unitaires en NodeJS avec Jest.

# Pourquoi tester ?

Les tests permettent simplement de s'assurer du bon fonctionnement et de la qualité de votre code. En effet, pour des projets aux fonctionnalités très simples, il peut sembler évident, juste en démarrant notre application et en l'utilisant manuellement, qu'elle fonctionne. Mais au fur et à mesure de son évolution vous allez ajouter des aspects de plus en plus complexes, et il est donc important de vous assurer que tout ce que vous avez implémenté continue de fonctionner correctement..

Les tests vous permettront également de prévenir les soucis de régression. Imaginons que votre code évolue et qu'une fonctionnalité soit altérée par une modification de votre code sans que vous ne vous en rendiez compte. Les test automatisés eux passeront sur l'entièreté de votre code et réussiront à détecter cette erreur pour vous la signaler.

## Les types de tests

Dans le vocabulaire professionnel, vous pouvez parfois entendre différents types de tests, dont voici une liste non exhaustive :

### Les Tests unitaires - Unit Tests

Ils ont pour but de tester des fonctions ou des composants individuels de manière isolée.

Par exemple : Vérifier que la fonction `add(2, 3)` retourne 5.

### Les Tests d'intégration - Integration Tests

Ils ont pour but de tester la combinaison de plusieurs unités ou modules pour s'assurer qu'ils fonctionnent ensemble.

Par exemple: Vérifier que l'API d'une base de données renvoie les données correctes.

### Les Tests E2E - End-to-End Tests

Ils ont pour but de simuler le comportement des utilisateurs dans un environnement réel pour tester un système complet.

Par exemple : Vérifier qu'un utilisateur peut s'inscrire sur un site web.

### Les Tests de performance - Performance Tests

Ils ont pour but de mesurer la rapidité et la réactivité du système.

Par exemple : Vérifier que le chargement d'une page ne dépasse pas 3 secondes.

### Les Tests d'accessibilité - Accessibility Tests

Ils ont pour but de s'assurer que le système est utilisable par tous, y compris les personnes en situation de handicap.



Par exemple : Vérifier que tous les boutons ont des [étiquettes ARIA](#) appropriées .

## Les Tests d'interface utilisateur - UI Tests

Ils ont pour but de vérifier que l'interface utilisateur s'affiche et fonctionne comme prévu.

Par exemple : S'assurer qu'une icône apparaît au bon endroit dans une application.

## Le framework Jest

Pour réaliser des tests en NodeJS nous allons donc utiliser le package `jest`, et `ts-jest` pour les tests en typescript. Le package `@types/jest` est aussi nécessaire pour les tests en typescript.

Pour l'installation (même si je suis convaincu que vous l'auriez trouvé par vous même ^\_^) :

```
$ npm install -D jest ts-jest @types/jest
```

## Configuration

Le fonctionnement de Jest est basé sur le fichier de configuration `jest.config.ts` qui contiendra sa configuration pour fonctionner avec TypeScript :

```
import type { Config } from 'jest';

const config: Config = {
  testEnvironment: 'node',
  transform: {
    '^.+\\.tsx?$': ['ts-jest', {}],
  },
  verbose: true
};

export default config;
```

Nous allons ensuite ajuster notre `package.json` en précisant le script test

```
[...]
"scripts": {
  [...]
  "test": "echo \"Error: no test specified\" && exit 1"
},
[...]
```

```
[...]
"scripts": {
  [...]
```

```
"test": "jest"
},
[...]
```

De cette manière, le lancement de ce script (via `npm run test`) cherchera l'ensemble des fichiers `*.spec.js` ou `*.test.js`.

Par convention, il est recommandé de créer un fichier `*.test.js` pour chaque fichier que vous possédez, et de les stocker dans un répertoire dédié aux tests. Par exemple si nous avons un fichier `math.js` qui assumera des fonctions mathématique, nous créerons son fichier de test correspondant `math.test.js`

## Nos premiers tests unitaires

Nous allons donc commencer avec des tests unitaires très simples. Nous allons essayer de tester une fonction `sum` contenue dans le fichier `math.ts` comme celle que nous avons définie plus tôt dans ce cours.

Pour cela, nous allons créer un fichier `tests/math.test.js` et ajoutez y les lignes suivantes:

```
//import { add } from '../src/math';
const add = require('../src/math');

describe('add function', () => {
  test('adds 1 + 2 to equal 3', () => {
    const res = add(1, 2);
    expect(res).toBe(3);
  });
});
```

Il fait référence au fichier `math.js` qui contient notamment notre fonction `add`

```
function add(a: number, b: number): number {
  //function add(a, b) {
    return a + b;
  }
  module.exports = add;
```

Vous remarquerez que la syntaxe de jest est plutôt explicite ce qui permet de comprendre rapidement le rôle de chaque partie.

- Le mot-clé `test` indique que nous allons créer un test avec l'intitulé ('adds 1 + 2 to equal 3') et le second paramètre correspond à un callback (via la partie `() => {...}`) de ce que le test doit vérifier.
- `expect` correspond à l'attendu, qui sera comparé à l'aide du chaînage de la fonction `toBe` qui vérifie que `res`, c'est à dire le résultat de la fonction `add(1, 2)`, est bien égal à 3.

Ainsi, en lançant le script test (`npm run test`), le résultat devrait ressembler à ceci :

```
PS C:\Users\guerr\OneDrive\Bureau\NodeJS\kgcoursiut> npm run test

> kgcoursiut@1.0.0 test
> jest

PASS tests/math.test.js
  add function
    ✓ adds 1 + 2 to equal 3 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.899 s, estimated 3 s
Ran all test suites.
```

Vous trouverez la liste des fonctions de comparaison à l'adresse : <https://jestjs.io/docs/expect> .

## Hiérarchisation des tests

Supposons que nous voulons tester plusieurs fonctions différentes et que chaque fonction possède sa propre batterie de tests. C'est alors que rentre en jeu la fonction `describe` que nous n'avons pas encore détaillée, et qui permet tout simplement de regrouper des tests.

```
// import { add } from '../src/math';
const add = require('../src/math');

describe('add function', () => {
  test('adds 1 + 2 to equal 3', () => {
    const res = add(1, 2);
    expect(res).toBe(3);
  });

  test('add -1 + 0', () => {
    const res = add(-1, 0);
    expect(res).toBe(-1);
  });
});
```

*Remarque : Depuis le début nous utilisons le mot clé `test` pour définir un test. Il est possible de le remplacer par `it` qui est un alias. C'est d'ailleurs souvent préféré par de nombreux développeurs pour des raisons de lisibilité / longueur de code.*

## Coverage

Le coverage est une notion importante dans les tests. Il permet de savoir si l'ensemble de votre code est testé. Jest permet de générer un rapport de couverture de code pour savoir si l'ensemble de votre code est testé ou non sous forme de pourcentage ou alors d'un rapport détaillé ligne par ligne.

Pour ajouter le coverage dans vos tests, il suffit d'ajouter le suffixe `--coverage` à votre commande `jest --coverage`.

Vous trouverez le résultat du coverage dans le dossier `coverage/index.html`.

## Seuil de couverture

Vous pouvez définir un seuil de couverture pour vos tests. S'il n'est pas atteint, les tests seront considérés comme échoués. On le définit dans le fichier `jest.config.js` avec la clé `coverageThreshold`. Par exemple, pour définir un seuil de 90% pour chaque type de couverture (branches, fonctions, lines, statements), ajoutez le code suivant :

```
import type { Config } from 'jest';

const config: Config = {
  [...]
  coverageThreshold: {
    global: {
      branches: 90,
      functions: 90,
      lines: 90,
      statements: 90,
    },
  },
  [...]
```

On appréciera généralement un seuil de couverture de 90% ou plus pour un projet en production.

## Environnement de test

Notre but est donc à présent de tester une API. Cependant un problème va se poser. En effet, si nous testons une API, nous allons devoir y faire appel et fatalement altérer les données de la base en les supprimant ou en les modifiant. Pour éviter cela il y a plusieurs méthodes:

- Utiliser une base de données de test.
- Utiliser un mock de la base de données, c'est-à-dire un outil qui va changer les retours des fonctions qui font appel à la base de données.
- Utiliser une base de données en mémoire, qui sera donc supprimée à la fin des tests.

Chacune de ces solutions est viable et peut dépendre d'un contexte. Comme nous voulons une solution simple et rapide nous allons préférer mocker la base de données.

## Mocking de prisma

Pour mocker la base de données nous allons utiliser le package `jest-mock-extended`. Ce package permet de mocker des fonctions et de les remplacer par des fonctions vides ou des fonctions qui renvoient des valeurs spécifiques.

Il faut préciser avant le démarrage des tests qu'il faut remplacer notre singleton 'client.ts' par le mock. Cela peut se faire à travers un fichier de setup tests/jest.setup.ts :

```
import { PrismaClient } from '@prisma/client';
import { mockDeep, mockReset, DeepMockProxy } from 'jest-mock-extended';
import prisma from '../src/client';

jest.mock('../src/client', () => ({
  __esModule: true,
  default: mockDeep<PrismaClient>(),
}));

beforeEach(() => {
  mockReset(prismaMock);
});

export const prismaMock = prisma as unknown as DeepMockProxy<PrismaClient>;
```

Et pour que ce fichier soit appelé, il faut modifier le fichier jest.config.ts pour lui indiquer où trouver ce fichier de setup :

```
import type { Config } from 'jest';

const config: Config = {
  [...]
  setupFilesAfterEnv: ['./tests/jest.setup.ts'],
  [...]
};
```

## Création de tests

Maintenant nous allons créer un test pour une route de notre API. Commençons par tester la route /users qui retourne tous les utilisateurs de notre base de données. Nous utilisons le package supertest pour tester notre API. Il permet de faire des requêtes HTTP sur notre serveur depuis les tests.

On ajoutera aussi le mock de prisma dans notre fichier de test. Et on modifie le retour de la fonction findMany grâce à la fonction mockResolvedValue.

```
import request from 'supertest';
import { app, stopServer } from '../src';
import { prismaMock } from './jest.setup';
```

```

import jwt from 'jsonwebtoken';

afterAll(() => {
  stopServer();
});

describe('GET /users', () => {
  it('should return an array of users', async () => {
    prismaMock.user.findMany.mockResolvedValue([
      { id: 1, name: 'Alice' },
      { id: 2, name: 'Bob' },
    ]);

    const response = await request(app).get('/users');

    expect(response.status).toBe(200);
    expect(response.body).toEqual([
      { id: 1, name: 'Alice' },
      { id: 2, name: 'Bob' },
    ]);
  });
});

```

# Documenter son API

Le but de cette section est de découvrir comment documenter une API avec Swagger.

## Swagger et OpenAPI

Tout d'abord, revenons sur une base primordiale : DOCUMENTEZ VOS CODES !!!!

La documentation de vos codes permet :

- de faciliter la compréhension, en ayant une explication claire de leur rôle, de la manière d'y parvenir, des particularités prises en considération, ...
- de réduire les erreurs, en vous permettant de prendre du recul sur la manière d'expliquer les lignes que vous avez écrites,
- d'améliorer la maintenabilité, en vous faisant gagner un temps considérable lorsque vous vous reprendrez dans ce projet dans quelques temps, mais également en permettant à d'autres développeurs d'intégrer facilement le projet.

Et bien pour les APIs, c'est encore plus important puisque son but premier est de fournir de l'interopérabilité. Il vous faut donc amener toutes les informations nécessaires pour permettre une bonne utilisation de votre API.

Pour cela, nous allons aborder Swagger, un ensemble d'outils et de spécifications qui permettent de documenter et de tester des APIs. Cet outil est basé sur [la spécification OpenAPI](#).

Son fonctionnement est basé sur :

- un format de fichier (généralement du YAML ou JSON) qui décrit toutes les routes, les paramètres, les réponses et autres aspects de l'API
- une interface utilisateur qui permet de visualiser la documentation de l'API et d'interagir avec elle directement via un navigateur.

Autre bonne nouvelle : il est possible d'intégrer Swagger dans une application Node.js avec Express relativement rapidement !

Pour l'installation, ça passe par la partie module avec le `middlewaresswagger-ui-express`, par `yamljs` pour charger et analyser les fichiers YAML, et bien évidemment par les types correspondants en dev :

```
$ npm install swagger-ui-express yamljs
$ npm install -D @types/swagger-ui-express @types/yamljs
```

## Le fichier de configuration swagger.yaml

C'est ce fichier qui a pour but de contenir la description de votre API. Il contient tout d'abord une entête avec les informations générales de l'API :

```
openapi: 3.0.0
info:
  title: Prisma User API
  version: 1.0.0
  description: API pour gérer les utilisateurs et leurs posts
servers:
  - url: http://localhost:3000
    description: Serveur local
paths:
```

S'ensuit l'ensemble des endpoints sous la directive paths, avec pour chacune la même structure, précisant pour chaque méthode (get / post / ...), un résumé (summary) ainsi que chacune des différentes réponses possibles (200, 204, ...), avec leurs détails (description, contenu, ...).

Par exemple pour la partie get de l'endpoint /users, alors on pourrait retrouver :

```
/users:
  get:
    summary: Récupère tous les utilisateurs
    responses:
      "200":
        description: Liste des utilisateurs avec leurs posts
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: "#/components/schemas/UserWithPosts"
      "204":
        description: Aucun utilisateur trouvé
```

Vous avez compris que la structure même de ce fichier va être relativement redondante, et que vous serez donc très vite à l'aise pour manipuler son contenu.

De plus, c'est dans ce genre de travail d'automatisation qu'une IA peut vous être très utile. En effet, elle pourra générer automatiquement ce genre de fichier à partir du code source de l'API.



Pour la méthode post de ce même endpoint /user, on continue donc de la même manière :

```
post:
  summary: Crée un nouvel utilisateur
  security:
    - bearerAuth: []
  requestBody:
    required: true
    content:
      application/json:
        schema:
          type: object
          properties:
            name:
              type: string
              example: John Doe
  responses:
    "201":
      description: Utilisateur créé
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/User"
    "401":
      description: Non autorisé (token manquant ou invalide)
      content:
        application/json:
          schema:
            type: object
            properties:
              error:
                type: string
                example: Unauthorized
```

et une fois que nous avons abordé toutes les méthodes disponibles pour un endpoint, alors on passe au suivant et on recommence !

```
/users/{userId}:
  get:
    summary: Récupère un utilisateur par son ID
    parameters:
      - name: userId
        in: path
        required: true
        schema:
          type: integer
        description: ID de l'utilisateur
    responses:
      "200":
        description: Utilisateur trouvé avec ses posts
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/UserWithPosts"
      "404":
        description: Utilisateur non trouvé
        content:
          text/plain:
            schema:
              type: string
            example: Utilisateur non trouvé
```

Une fois tous les endpoints terminés, on va préciser les composants qui sont utilisés par l'API, c'est-à-dire les informations relatives à la sécurité,

```
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

ainsi que les schémas des entités manipulées.

```
schemas:
  User:
    type: object
    properties:
      id:
        type: integer
        example: 1
      name:
        type: string
        example: John Doe
  Post:
    type: object
    properties:
      id:
        type: integer
        example: 1
      title:
        type: string
        example: My First Post
      authorId:
        type: integer
        example: 1
  UserWithPosts:
    allOf:
      - $ref: "#/components/schemas/User"
      - type: object
        properties:
          posts:
            type: array
            items:
              $ref: "#/components/schemas/Post"
```

## Configuration de Swagger dans l'application avec Express

Si nous reprenons notre fichier principal `index.ts`, alors nous allons devoir intégrer quelques éléments pour que la documentation puisse être affichée :

Dans l'entête :

```
import express from 'express';
import { userRouter } from './user/user.router';
// On ajoute les informations pour swagger
import swaggerUi from 'swagger-ui-express';
import YAML from 'yamljs';
import path from 'path';
[...]
```

et une fois le serveur lancé, avant de charger les middleware des différents endpoints :

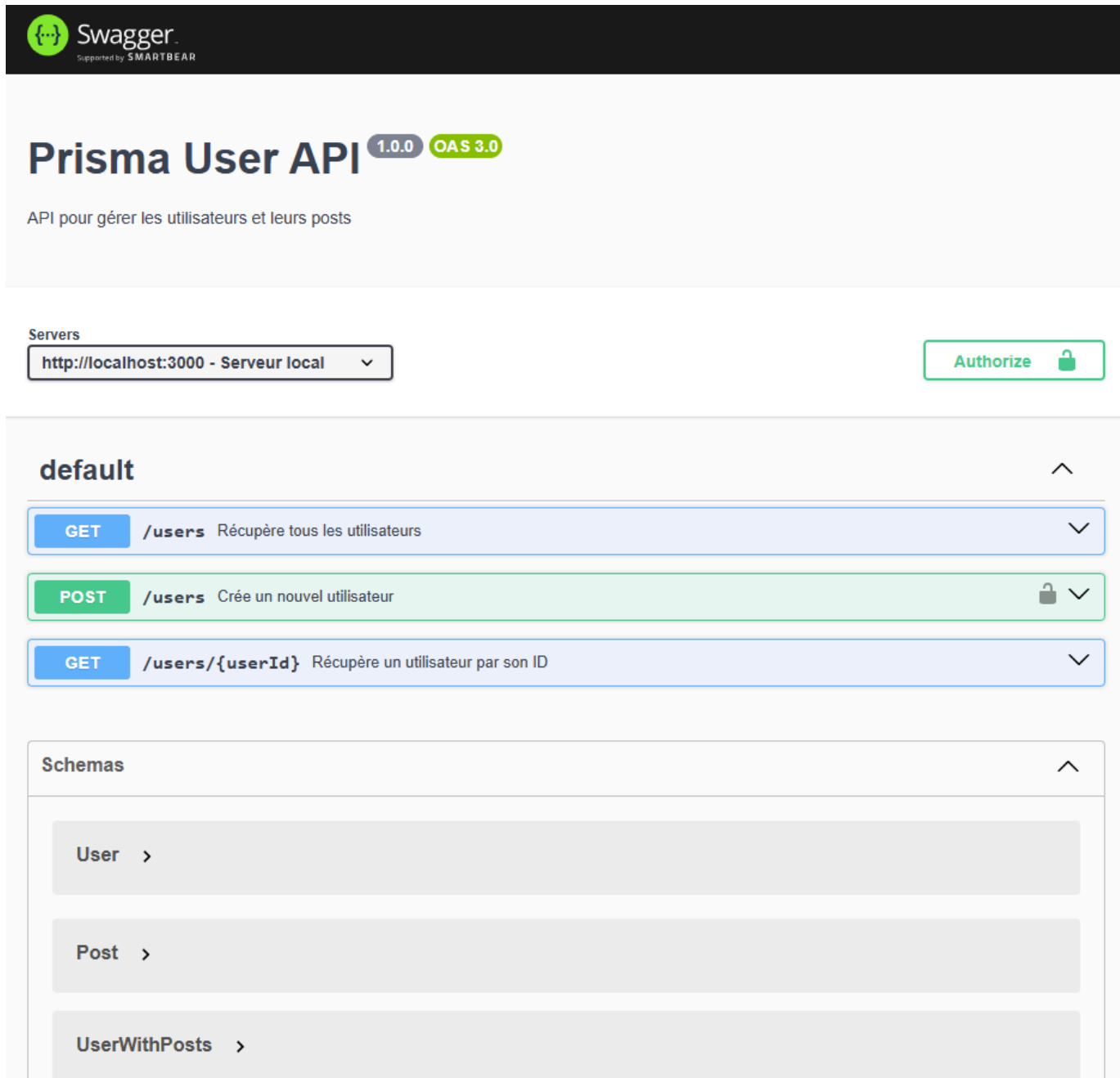
```
[...]
export const app = express();
const port = 3000;
app.listen(port, () => {
  console.log(`Mon serveur démarre sur le port ${port}`);
});

// On charge la spécification Swagger
const swaggerDocument = YAML.load(path.join(__dirname, './swagger.yaml'));
// Et on affecte le Serveur Swagger UI à l'adresse /api-docs
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
[...]
```

Le reste, vous connaissez ;-)

## Tester l'API et la documentation

Une fois votre application lancée, vous pouvez accéder à la documentation Swagger UI en naviguant à l'adresse <http://localhost:3000/api-docs> dans votre navigateur.



Swagger  
Supported by SMARTBEAR

# Prisma User API 1.0.0 OAS 3.0

API pour gérer les utilisateurs et leurs posts

Servers

[http://localhost:3000 - Serveur local](#) [Authorize](#)

## default

- GET** `/users` Récupère tous les utilisateurs
- POST** `/users` Crée un nouvel utilisateur
- GET** `/users/{userId}` Récupère un utilisateur par son ID

## Schemas

- User >
- Post >
- UserWithPosts >

Vous y trouverez toutes les informations sur vos endpoints, et vous pourrez tester les différentes requêtes API directement à partir de l'interface Swagger.