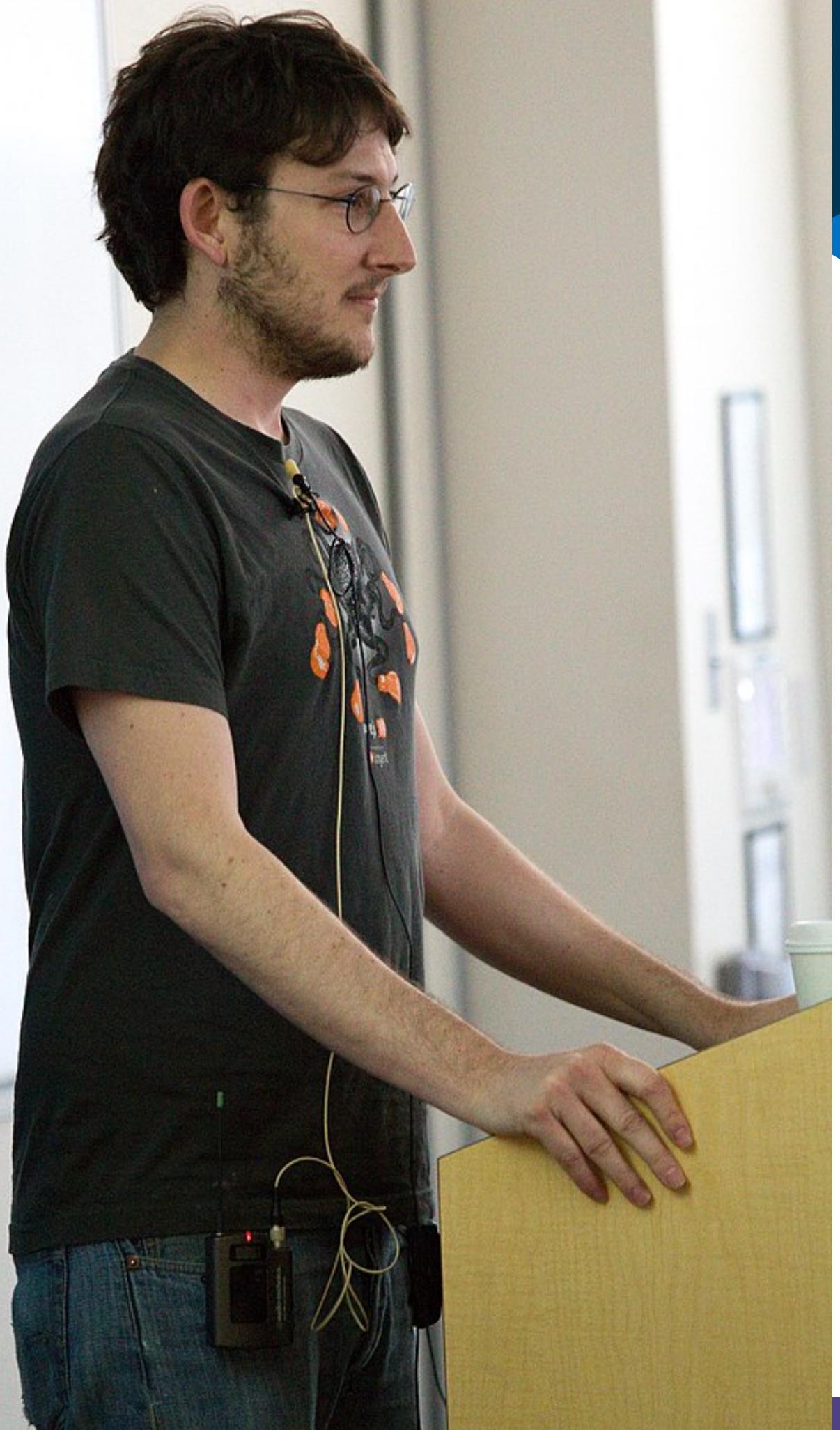


# FORMATION NODE.JS

Pierrick Hauguel

COURGETTE  
&  
GALIPETTE



# Historique rapide

---

Node.js a une histoire intéressante. À l'origine, Ryan Dahl travaillait sur une application de chat en ligne qui utilisait Ruby on Rails, mais il a rencontré des problèmes de performances lorsqu'il a essayé de faire fonctionner l'application en temps réel. Il a donc décidé de créer Node.js en 2009, avec pour objectif de résoudre ces problèmes de performance en utilisant le moteur JavaScript V8 de Google.

Depuis sa création, Node.js a connu une croissance rapide, avec une communauté de développeurs en constante expansion et de nombreuses applications à succès. Par exemple, PayPal a adopté Node.js en 2013, ce qui a permis de réduire le temps de chargement de ses pages de 200 à 40 millisecondes, entraînant une augmentation de 2% du taux de conversion.



# Panorama rapide

---

Node.js utilise le moteur JavaScript V8 de Google, qui est également utilisé dans le navigateur Google Chrome. Cela signifie que les développeurs peuvent écrire du code en utilisant le même langage des deux côtés du serveur, ce qui facilite la création d'applications Web en temps réel.

L'un des exemples les plus courants d'applications Web en temps réel est le chat en ligne. En utilisant Node.js, les développeurs peuvent créer des applications de chat en temps réel qui sont rapides et réactives, même lorsqu'elles sont utilisées par un grand nombre de personnes en même temps. Cela est possible grâce à l'utilisation de sockets Web, qui permettent des connexions bi-directionnelles entre le client et le serveur.



# Fonctionnement interne

---

Node.js utilise un modèle d'E/S non bloquante et asynchrone, ce qui signifie que les opérations d'entrée/sortie ne bloquent pas le thread principal, ce qui permet d'optimiser les performances de l'application.

Un exemple concret de cela est l'opération de lecture de fichiers. Avec d'autres langages de programmation, une opération de lecture de fichiers peut bloquer le thread principal, ce qui entraîne des temps de latence importants. Avec Node.js, cependant, l'opération de lecture de fichiers est effectuée de manière asynchrone, ce qui permet au thread principal de continuer à traiter d'autres requêtes pendant que l'opération de lecture de fichiers est en cours.

Cela permet à Node.js de gérer efficacement un grand nombre de connexions en temps réel, comme dans le cas des applications de chat.



# Exemples d'applications :

---

- Applications de chat en temps réel : Twitch, Slack, et de nombreux autres sites Web utilisent Node.js pour gérer les connexions en temps réel et permettre aux utilisateurs d'interagir sans avoir à actualiser constamment la page.
- Applications de streaming : Netflix utilise Node.js pour son système de streaming vidéo, qui permet une lecture fluide des vidéos pour des millions d'utilisateurs dans le monde entier.
- Applications de traitement d'images : Des bibliothèques telles que Sharp permettent de redimensionner et de compresser des images de manière efficace, ce qui est utile pour les sites de commerce électronique tels que Zalando et Redbubble.
- Applications de l'Internet des objets (IoT) : Node.js peut être utilisé pour programmer des capteurs intelligents, des appareils de maison intelligents et des objets connectés. Un exemple est le système de maison intelligente Home Assistant, qui utilise Node.js pour gérer les périphériques connectés à la maison.



# Avantages et inconvénients :

---

Les avantages de Node.js incluent sa capacité à gérer de grandes quantités de connexions en temps réel grâce à son modèle d'E/S non bloquante et asynchrone, qui permet de traiter de nombreuses requêtes en même temps. Node.js utilise également JavaScript des deux côtés du serveur, ce qui permet aux développeurs de travailler avec un langage familier et d'éviter les problèmes de compatibilité.

En termes de mise à l'échelle horizontale, Node.js est facilement extensible et peut être déployé sur plusieurs serveurs pour gérer des charges de travail plus importantes. Cela en fait une option attrayante pour les grandes entreprises qui doivent traiter de grandes quantités de données en temps réel.



# Avantages et inconvénients :

---

Cependant, l'un des inconvénients de Node.js est la courbe d'apprentissage abrupte pour les développeurs qui ne sont pas familiers avec JavaScript. Cela peut rendre la programmation en Node.js plus difficile pour les nouveaux développeurs. De plus, il est important de gérer correctement les modules tiers en utilisant des pratiques de sécurité appropriées pour éviter les vulnérabilités de sécurité potentielles.



# Rappels ES6

---

```
let x = 1;
if (true) {
  let x = 2; // La portée de cette variable x est limitée au bloc if
  console.log(x); // Affiche 2
}
console.log(x); // Affiche 1
```



# Les fonctions fléchées

---

Les fonctions fléchées permettent de définir des fonctions de manière plus concise et expressive. Voici un exemple de code utilisant des fonctions fléchées :

```
const nums = [1, 2, 3];
const doubled = nums.map(num => num * 2);
console.log(doubled); // Affiche [2, 4, 6]
```



# Les paramètres par défaut

---

Les paramètres par défaut permettent de définir des valeurs par défaut pour les arguments des fonctions. Voici un exemple de code utilisant des paramètres par défaut :

```
function sayHello(name = 'World') {  
  console.log(`Hello, ${name}!`);  
}  
  
sayHello(); // Affiche "Hello, World!"  
sayHello('John'); // Affiche "Hello, John!"
```



# Le reste et l'étalement

---

Le reste et l'étalement permettent de manipuler des tableaux et des objets de manière plus expressive. Voici un exemple de code utilisant le reste pour regrouper des arguments de fonction en un tableau :

```
function sum(...nums) {  
  return nums.reduce((acc, curr) => acc + curr);  
}  
  
console.log(sum(1, 2, 3, 4)); // Affiche 10
```



# Les promesses

---

Les promesses permettent de gérer les opérations asynchrones de manière plus facile et plus expressive. Voici un exemple de code utilisant des promesses :

```
function getData() {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      const data = [1, 2, 3];  
      resolve(data);  
    }, 1000);  
  });  
}  
getData().then(data => console.log(data)); // Affiche [1, 2, 3] après une s
```



# Le destructuring

---

Le destructuring permet de décomposer des tableaux et des objets en variables individuelles, ce qui facilite la manipulation des données. Voici un exemple de code utilisant le destructuring :

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // Affiche "John"
console.log(age); // Affiche 30
```



# Les itérateurs et les itérables

---

Le destructuring permet de décomposer des tableaux et des objets en variables individuelles, ce qui facilite la manipulation des données. Voici un exemple de code utilisant le destructuring :

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // Affiche "John"
console.log(age); // Affiche 30
```



# Installation et découverte :

Node.js est disponible en téléchargement sur le site officiel (<https://nodejs.org/>). Une fois téléchargé, l'installation de Node.js est généralement simple et rapide. Il suffit de suivre les instructions d'installation pour votre système d'exploitation préféré.

Une fois que Node.js est installé, vous pouvez commencer à écrire des applications en utilisant l'API Node.js. L'API de Node.js comprend un ensemble de modules qui peuvent être utilisés pour effectuer diverses tâches, telles que la gestion des fichiers, la création d'un serveur Web, la communication en réseau, et bien plus encore.



# Installation et découverte :

---

Il existe également un grand nombre de packages et de modules tiers disponibles pour Node.js, qui peuvent être facilement installés à l'aide du gestionnaire de packages npm (Node Package Manager). Cela permet aux développeurs d'ajouter rapidement et facilement des fonctionnalités supplémentaires à leurs applications en utilisant des modules tiers existants.



# Application "hello world" :

Pour commencer à utiliser Node.js, voici un exemple simple d'application "hello world" :

```
// Chargement du module http
const http = require('http');

// Création d'un serveur
const server = http.createServer((req, res) => {
    // Envoi d'une réponse
    res.end('Hello World!');
});

// Démarrage du serveur
server.listen(3000, () => {
    console.log('Le serveur est démarré sur le port 3000');
});
```

Ce code crée un serveur Web qui renvoie une réponse "Hello World!" à toutes les requêtes entrantes. L'application écoute sur le port 3000, ce qui signifie qu'elle peut être testée en accédant à l'URL <http://localhost:3000> dans un navigateur Web.



# Les modules

---

Les modules sont des blocs de code réutilisables qui permettent aux développeurs de séparer leur code en différentes parties, ce qui facilite la maintenance et le développement de leurs applications. Node.js dispose d'un certain nombre de modules de base, ainsi que d'un grand nombre de modules tiers disponibles via le gestionnaire de packages npm.

Se servir des modules de base : Node.js dispose d'un certain nombre de modules de base intégrés qui peuvent être utilisés pour effectuer des tâches courantes telles que la gestion des fichiers, la création de serveurs Web, l'interaction avec des bases de données, et bien plus encore. Les modules de base sont inclus automatiquement dans l'installation de Node.js, ce qui signifie qu'ils peuvent être utilisés immédiatement sans avoir besoin d'installer quoi que ce soit d'autre.



# Les modules

---

Par exemple, le module `fs` peut être utilisé pour interagir avec le système de fichiers, en lisant et en écrivant des fichiers. Le module `http` peut être utilisé pour créer un serveur Web et écouter des requêtes entrantes. Les modules `crypto` et `zlib` peuvent être utilisés pour effectuer des opérations de chiffrement et de décompression, respectivement.



# Les modules

---

Par exemple, le module `fs` peut être utilisé pour interagir avec le système de fichiers, en lisant et en écrivant des fichiers. Le module `http` peut être utilisé pour créer un serveur Web et écouter des requêtes entrantes. Les modules `crypto` et `zlib` peuvent être utilisés pour effectuer des opérations de chiffrement et de décompression, respectivement.



# Les informations système avec le module os :

Le module os de Node.js fournit un certain nombre de fonctions pour accéder aux informations du système. Voici quelques exemples :

Récupération du nom de l'utilisateur actuel :

```
const os = require('os');

console.log(`Utilisateur actuel : ${os.userInfo().username}`);
```

Récupération de l'architecture du processeur :

```
const os = require('os');

console.log(`Architecture du processeur : ${os.arch()}`);
```



# Les informations système avec le module os :

Récupération de la quantité de mémoire disponible :

```
const os = require('os');

const totalMemory = os.totalmem();
const freeMemory = os.freemem();

console.log(`Mémoire totale : ${totalMemory}`);
console.log(`Mémoire libre : ${freeMemory}`);
```

Récupération de la version du système d'exploitation :

```
const os = require('os');

console.log(`Version du système d'exploitation : ${os.version()}`);
```



# Les informations système avec le module os :

Récupération de l'adresse IP du système :

```
const os = require('os');

const networkInterfaces = os.networkInterfaces();

const ipAddresses = networkInterfaces['eth0']
  .filter((interface) => interface.family === 'IPv4')
  .map((interface) => interface.address);

console.log(`Adresses IP : ${ipAddresses}`);
```

Récupération des informations sur les processeurs :

```
const os = require('os');

const cpuInfo = os.cpus();

console.log(`Nombre de processeurs : ${cpuInfo.length}`);
console.log(`Vitesse de l'horloge : ${cpuInfo[0].speed}`);
console.log(`Modèle du processeur : ${cpuInfo[0].model}`);
```



# Les informations système avec le module os :

Récupération des informations sur les interfaces réseau :

```
const os = require('os');

const networkInterfaces = os.networkInterfaces();

Object.keys(networkInterfaces).forEach((interfaceName) => {
    const interfaceInfo = networkInterfaces[interfaceName];

    console.log(`Interface : ${interfaceName}`);

    interfaceInfo.forEach((info) => {
        console.log(`    Adresse IP : ${info.address}`);
        console.log(`    Masque de sous-réseau : ${info.netmask}`);
    });
});
```



# NPM

---

npm (Node Package Manager) est le gestionnaire de packages officiel de Node.js. Il permet aux développeurs de télécharger, installer et gérer des packages tiers pour leurs applications Node.js. npm est livré avec Node.js, ce qui signifie qu'il est facilement accessible une fois que Node.js est installé.

npm comprend un grand nombre de packages tiers disponibles, qui peuvent être facilement installés et utilisés dans les applications Node.js. Les développeurs peuvent rechercher des packages spécifiques à l'aide de la commande `npm search`, installer des packages en utilisant la commande `npm install`, et gérer les packages installés à l'aide de la commande `npm update`. npm est un outil essentiel pour les développeurs Node.js qui cherchent à ajouter rapidement et facilement des fonctionnalités à leurs applications.



# Serveur Web

Node.js est un excellent choix pour créer des serveurs Web en raison de sa capacité à gérer de grandes quantités de connexions en temps réel et de sa facilité d'utilisation. Voici quelques éléments clés pour la création de serveurs Web en Node.js.

Ma première application web : Pour créer une application Web de base en Node.js, vous pouvez utiliser le module http intégré pour créer un serveur et écouter les requêtes entrantes. Voici un exemple simple qui renvoie une réponse "Hello World!" à toutes les requêtes entrantes :

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.end('Hello World!');
});

server.listen(3000, () => {
  console.log('Le serveur est démarré sur le port 3000');
});
```



# Gérer les requêtes et les réponses

## HTTP :

Le serveur Web créé précédemment utilise la méthode `createServer` du module `http` pour écouter les requêtes HTTP entrantes. La fonction de rappel passée à `createServer` est appelée chaque fois qu'une nouvelle requête HTTP est reçue. Cette fonction prend deux arguments : une demande (objet de type `http.IncomingMessage`) qui contient des informations sur la requête entrante, et une réponse (objet de type `http.ServerResponse`) qui est utilisée pour envoyer la réponse au client.



# Routage des URLs :

---

Le routage des URLs est une technique courante utilisée pour déterminer quelle action doit être effectuée en fonction de l'URL demandée par le client. Dans Node.js, le routage peut être implémenté en utilisant un module tiers tel que express, ou en utilisant les modules de base tels que http et url.

```
const http = require('http');
const url = require('url');

const server = http.createServer((request, response) => {
  const path = url.parse(request.url).pathname;

  if (path === '/') {
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.write('Bienvenue sur la page d\'accueil !');
    response.end();
  } else if (path === '/about') {
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.write('À propos de moi');
    response.end();
  } else {
    response.writeHead(404, { 'Content-Type': 'text/plain' });
    response.write('Page non trouvée');
    response.end();
  }
});

server.listen(3000);
```



## Les modules fs et path :

Les modules fs et path sont deux modules de base de Node.js qui peuvent être utilisés pour effectuer des opérations sur le système de fichiers. Le module fs fournit des fonctions pour lire et écrire des fichiers, créer des répertoires, et bien plus encore. Le module path fournit des fonctions pour travailler avec des chemins de fichiers et de répertoires.

Voici un exemple d'utilisation du module fs pour lire le contenu d'un fichier et l'envoyer en réponse à une requête HTTP :



# Les modules fs et path :

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((request, response) => {
  const filePath = './public/index.html';

  fs.readFile(filePath, (err, content) => {
    if (err) {
      response.writeHead(500, { 'Content-Type': 'text/plain' });
      response.write(`Une erreur est survenue : ${err.message}`);
      response.end();
    } else {
      response.writeHead(200, { 'Content-Type': 'text/html' });
      response.write(content);
      response.end();
    }
  });
});

server.listen(3000);
```



# Opérations bloquantes et non-bloquantes :

Les opérations bloquantes sont des opérations qui bloquent le thread principal de Node.js jusqu'à ce qu'elles soient terminées. Les opérations non-bloquantes sont des opérations qui ne bloquent pas le thread principal et sont exécutées en arrière-plan.

Il est important d'éviter les opérations bloquantes dans une application Node.js, car elles peuvent ralentir l'ensemble de l'application. Les opérations non-bloquantes, en revanche, permettent à l'application de continuer à répondre aux requêtes HTTP entrantes tout en effectuant des tâches en arrière-plan.



# Opérations bloquantes et non-bloquantes :

Un exemple d'opération bloquante courante est la lecture synchronisée d'un fichier à l'aide de la méthode `fs.readFileSync`. Cette méthode bloque le thread principal de Node.js jusqu'à ce que la lecture du fichier soit terminée, ce qui peut entraîner un ralentissement de l'ensemble de l'application si elle est utilisée pour traiter des requêtes HTTP entrantes. Voici un exemple :

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((request, response) => {
  const filePath = './public/index.html';

  try {
    const content = fs.readFileSync(filePath);
    response.writeHead(200, { 'Content-Type': 'text/html' });
    response.write(content);
    response.end();
  } catch (err) {
    response.writeHead(500, { 'Content-Type': 'text/plain' });
    response.write(`Une erreur est survenue : ${err.message}`);
    response.end();
  }
});

server.listen(3000);
```



# Découverte d'Express :

Express est un framework Web pour Node.js qui fournit des fonctionnalités utiles pour la création d'applications Web telles que le routage, la gestion des cookies et des sessions, et la gestion des erreurs. Express est populaire pour sa flexibilité et sa facilité d'utilisation. Voici un exemple d'application Express simple :

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

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

app.listen(3000, () => {
  console.log('Le serveur a démarré sur le port 3000');
});
```



# Génération du projet avec la CLI :

La CLI d'Express permet de générer rapidement un squelette de projet Express en quelques commandes. Pour générer un projet, il suffit d'installer la CLI d'Express via NPM, puis de lancer la commande `express nom-du-projet`. Cela générera une structure de dossier de projet de base, avec des fichiers et des dossiers préconfigurés pour l'utilisation d'Express. Voici un exemple de génération d'un projet Express à l'aide de la CLI :

```
$ npm install -g express-generator  
$ express nom-du-projet  
$ cd nom-du-projet  
$ npm install  
$ npm start
```



# Architectures

---

L'architecture du code d'une application Express dépendra de nombreux facteurs, tels que la taille de l'application, la complexité des fonctionnalités, et plus encore. Cependant, il est courant de diviser l'application en modules distincts pour les différentes fonctionnalités, tels que les routes, les contrôleurs et les modèles.



# Architecture simple

```
- app.js  
- routes/  
  - index.js  
  - users.js  
- controllers/  
  - userController.js  
- models/  
  - userModel.js  
- views/  
  - index.ejs  
  - login.ejs
```

L'application Express est généralement divisée en plusieurs modules, tels que les routes, les contrôleurs et les modèles. Les fichiers de routes définissent les routes de l'application et font appel aux contrôleurs pour effectuer les actions appropriées. Les contrôleurs contiennent la logique métier de l'application et interagissent avec les modèles pour récupérer et stocker des données. Les modèles représentent les données de l'application et contiennent des méthodes pour interagir avec la base de données. Les fichiers de vues définissent l'interface utilisateur de l'application en utilisant un moteur de template tel que EJS.



## Exporter des modules avec module.exports

Le mécanisme d'export de base dans Node.js est l'utilisation de la propriété `module.exports`. Voici un exemple de code exportant une fonction depuis un module :

Dans le fichier `module.js` :

```
function sayHello(name) {  
  console.log(`Hello, ${name}!`);  
}  
  
module.exports = sayHello;
```



## Exporter plusieurs éléments avec exports

Node.js fournit également un objet exports qui peut être utilisé pour exporter plusieurs éléments en même temps. Voici un exemple de code exportant plusieurs fonctions depuis un module :

Dans le fichier module.js :

```
exports.sayHello = function(name) {  
    console.log(`Hello, ${name}!`);  
};  
  
exports.sayGoodbye = function(name) {  
    console.log(`Goodbye, ${name}!`);  
};
```



## Exporter des éléments avec des noms personnalisés

Dans certains cas, vous pouvez souhaiter exporter des éléments avec des noms personnalisés, plutôt que d'utiliser les noms de fonction ou de variable d'origine. Pour cela, vous pouvez utiliser l'objet exports avec des noms de propriété personnalisés. Voici un exemple de code exportant une fonction avec un nom personnalisé depuis un module :

Dans le fichier module.js :

```
function sayHello(name) {  
    console.log(`Hello, ${name}!`);  
}  
  
exports.greeting = sayHello;
```

# Requêtes HTTP

Express fournit une API simple pour gérer les requêtes HTTP entrantes dans une application web. Les routes Express peuvent être utilisées pour mapper les demandes HTTP à des fonctions de contrôleur qui sont chargées de traiter la demande et de renvoyer une réponse.

Voici un exemple de route pour gérer une demande GET :

```
app.get('/users', function(req, res) {  
  res.send('Liste des utilisateurs');  
});
```

Cette route répondra à une demande GET pour l'URL "/users" en renvoyant la chaîne "Liste des utilisateurs".



# Middlewares :

Les middlewares sont une fonctionnalité importante d'Express qui permettent de traiter les requêtes et les réponses HTTP. Les middlewares sont des fonctions qui sont exécutées à chaque demande entrante avant que la route appropriée ne soit exécutée.

Voici un exemple de middleware pour enregistrer les requêtes HTTP dans un fichier journal :

```
app.use(function(req, res, next) {
  console.log(` ${req.method} ${req.path} - ${req.ip}`);
  next();
});
```

Cette fonction enregistre les détails de la demande (méthode, chemin et adresse IP) dans la console à chaque fois qu'une demande est reçue.



# Gestion des erreurs :

Express fournit également un moyen de gérer les erreurs dans une application web. Les fonctions de gestion des erreurs sont des middlewares spéciaux qui sont exécutés lorsque des erreurs se produisent pendant le traitement de la demande.

Voici un exemple de gestionnaire d'erreurs pour renvoyer une erreur 404 lorsque la page demandée n'existe pas :

```
app.use(function(req, res, next) {  
  res.status(404).send('Page non trouvée');  
});
```



# Les sessions

---

Les sessions sont un mécanisme pour stocker des données de session entre les demandes HTTP. Express fournit un middleware pour la gestion des sessions appelé "express-session".

Voici un exemple de middleware de session pour stocker une variable d'utilisateur connecté entre les demandes :

```
const session = require('express-session');

app.use(session({
  secret: 'supersecret',
  resave: false,
  saveUninitialized: true
}));

app.get('/profil', function(req, res) {
  const utilisateur = req.session.utilisateur;
  res.send(`Profil de ${utilisateur}`);
});
```



# Body Parser

---

Body Parser est un middleware pour Express qui permet de récupérer les données envoyées dans le corps d'une requête HTTP.

```
npm install body-parser
```

Une fois que Body Parser est installé, vous pouvez l'utiliser en l'important dans votre fichier principal de l'application :

```
const bodyParser = require('body-parser');
```

Vous pouvez ensuite utiliser Body Parser en tant que middleware dans votre application Express :

```
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
```

Vous pouvez maintenant utiliser req.body pour accéder aux données envoyées dans le corps de la requête.



# Introduction aux moteurs de template

---

- Qu'est-ce qu'un moteur de template ?
  - Un moteur de template est un outil qui permet de générer du HTML à partir de modèles.
- Pourquoi utiliser des moteurs de template en Node.js ?
  - Les moteurs de template facilitent la génération dynamique de contenu HTML pour les applications Web.
- Quels sont les moteurs de template populaires pour Node.js ?
  - Les moteurs de template populaires pour Node.js sont EJS, Pug et Handlebars.



# Introduction à EJS

---

- Qu'est-ce que EJS ?
  - EJS est un moteur de template JavaScript simple et flexible pour Node.js.
- Pourquoi utiliser EJS ?
  - EJS est facile à utiliser et à comprendre.
  - EJS prend en charge la génération dynamique de contenu HTML pour les applications Web.
- Comment installer EJS ?
  - `npm install ejs`



# Création et utilisation de vues avec EJS

EJS peut être utilisé pour créer des vues dynamiques en générant du HTML à partir de données côté serveur. Voici comment créer une vue avec EJS :

1. Créez un fichier de vue avec l'extension .ejs, par exemple `hello.ejs`.
2. Ajoutez du contenu HTML au fichier de vue, en utilisant les balises EJS pour les éléments dynamiques, par exemple :

```
<html>
  <head>
    <title><%= pageTitle %></title>
  </head>
  <body>
    <h1><%= message %></h1>
  </body>
</html>
```



# Création et utilisation de vues avec EJS

---

Dans votre application Express, utilisez la fonction `res.render()` pour envoyer la vue au client, en passant les données nécessaires en deuxième argument, par exemple :

```
app.get('/', function(req, res) {
  res.render('hello', {
    pageTitle: 'Page d\'accueil',
    message: 'Bonjour, monde!'
  });
});
```



# Les boucles

---

EJS prend en charge l'utilisation de structures de contrôle telles que les boucles et les conditions dans les vues, ce qui permet de générer des vues dynamiques plus complexes. Voici un exemple d'utilisation d'une boucle for dans une vue EJS :

```
<ul>
  <% for (let i = 0; i < users.length; i++) { %>
    <li><%= users[i].name %></li>
  <% } %>
</ul>
```



Le code ci-dessus générera une liste d'utilisateurs en utilisant une boucle for pour parcourir le tableau users, qui contient des objets utilisateur avec une propriété name. La balise EJS `<%= %>` est utilisée pour afficher le nom de chaque utilisateur dans la liste.



# Les partials

---

EJS prend également en charge l'utilisation de partials pour inclure du contenu réutilisable dans plusieurs vues. Voici un exemple d'utilisation de partials dans une vue EJS :

```
<html>
  <head>
    <title><%= pageTitle %></title>
  </head>
  <body>
    <% include('header') %>
    <h1><%= message %></h1>
    <% include('footer') %>
  </body>
</html>
```

Le code ci-dessus inclut deux partials dans la vue principale : header.ejs et footer.ejs. Les partials peuvent contenir du code HTML réutilisable, tel que des en-têtes de page, des pieds de page ou des menus de navigation. La balise EJS `<% include('partial') %>` est utilisée pour inclure le contenu du partial dans la vue principale.



# Les layout

---

EJS permet également d'étendre des fichiers de layout pour créer des pages plus complexes. Il est possible d'inclure du contenu statique dans un layout, comme les en-têtes et les pieds de page, et de remplir le contenu dynamique à l'aide de la vue associée.

Voici un exemple de layout EJS :

```
<!doctype html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <header>
      <% include('partials/header') %>
    </header>
    <main>
      <%- body %>
    </main>
    <footer>
      <% include('partials/footer') %>
    </footer>
  </body>
</html>
```



# Présentation de MongoDB

---

MongoDB est un système de gestion de base de données orienté document, qui stocke des données sous forme de documents BSON (Binary JSON). Les documents peuvent contenir des données structurées et non structurées, et peuvent être indexés pour des recherches rapides.

MongoDB est particulièrement adapté aux applications Web modernes, car il offre une grande flexibilité pour stocker des données dans un format JSON-like, qui est facilement manipulable avec JavaScript.

Mongoose est une bibliothèque ODM (Object Data Modeling) pour MongoDB, qui offre une interface simple et élégante pour interagir avec la base de données. Il peut être installé via NPM avec la commande `npm install mongoose`.



# Configuration et connexion à la BDD

Une fois que MongoDB et Mongoose sont installés, la connexion à la base de données peut être établie en utilisant la fonction `mongoose.connect()`. Cette fonction accepte une chaîne de connexion qui spécifie l'adresse de la base de données.

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/mydatabase');
```



# Créer des schémas de données

Mongoose utilise des schémas de données pour définir la structure des documents qui seront stockés dans la base de données. Un schéma est une définition des champs de document, qui peuvent inclure le type de données, les validations et les options de configuration.

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const userSchema = new Schema({
  firstName: { type: String, required: true },
  lastName: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 18 }
});
```



# Insérer un document

Une fois qu'un schéma de données est défini, un document peut être créé en utilisant le modèle associé. Le modèle est créé en utilisant la fonction `mongoose.model()` et le schéma de données.

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const userSchema = new Schema({
  firstName: { type: String, required: true },
  lastName: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 18 }
});

const User = mongoose.model('User', userSchema);

const newUser = new User({
  firstName: 'John',
  lastName: 'Doe',
  email: 'john.doe@example.com',
  age: 30
});

newUser.save(function (err) {
  if (err) return handleError(err);
  console.log('User saved successfully!');
});
```



# Récupérer un ou plusieurs documents :

Une fois que les schémas sont créés et les documents sont insérés dans la base de données, il est possible de récupérer un ou plusieurs documents. Pour cela, on peut utiliser des méthodes de requête de Mongoose telles que `find()`, `findOne()` ou `findById()`.

Exemple de code pour récupérer tous les documents d'une collection :

```
const User = require('./models/user');

User.find({}, (err, users) => {
  if (err) {
    console.log(err);
  } else {
    console.log(users);
  }
});
```



# Récupérer un ou plusieurs documents :

Exemple de code pour récupérer un document spécifique en utilisant son ID :

```
const User = require('./models/user');

const userId = '123456789';

User.findById(userId, (err, user) => {
  if (err) {
    console.log(err);
  } else {
    console.log(user);
  }
});
```



# Modifier ou supprimer des documents :

Pour modifier ou supprimer un document existant dans la base de données, on peut utiliser les méthodes de mise à jour et de suppression de Mongoose telles que `updateOne()`, `updateMany()`, `deleteOne()` et `deleteMany()`. Exemple de code pour mettre à jour un document :

```
const User = require('./models/user');

const userId = '123456789';

User.updateOne({ _id: userId }, { name: 'John Doe' }, (err, result) => {
  if (err) {
    console.log(err);
  } else {
    console.log(result);
  }
});
```



# Modifier ou supprimer des documents :

Exemple de code pour supprimer un document :

```
const User = require('./models/user');

const userId = '123456789';

User.deleteOne({ _id: userId }, (err) => {
  if (err) {
    console.log(err);
  } else {
    console.log('Document deleted');
  }
});
```



# Exemple complet - express / ejs / mongodb

```
// Import des modules nécessaires
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const ejs = require('ejs');

// Initialisation de l'application Express
const app = express();

// Configuration de l'application Express
app.set('view engine', 'ejs');
app.use(bodyParser.urlencoded({ extended: true }));

// Connexion à la base de données MongoDB
mongoose.connect('mongodb://localhost:27017/myapp', { useNewUrlParser: true,
  .then(() => console.log('Connexion à la base de données réussie'))
  .catch((err) => console.log(`Erreur de connexion à la base de données : ${err}`))

// Création du schéma de données pour les utilisateurs
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number
});

// Création du modèle pour les utilisateurs
const User = mongoose.model('User', userSchema);
```



# Exemple complet - express / ejs / mongodb

```
// Définition de l'endpoint API pour récupérer tous les utilisateurs
app.get('/users', (req, res) => {
  User.find({}, (err, users) => {
    if (err) {
      console.log(err);
    } else {
      res.render('users', { users: users });
    }
  });
});

// Définition de l'endpoint API pour ajouter un utilisateur
app.post('/users', (req, res) => {
  const newUser = new User({
    name: req.body.name,
    email: req.body.email,
    age: req.body.age
  });
  newUser.save((err) => {
    if (err) {
      console.log(err);
    } else {
      res.redirect('/users');
    }
  });
});

// Démarrage du serveur
app.listen(3000, () => {
  console.log('Serveur démarré sur le port 3000');
})
```



# Exemple complet - express / ejs / mongodb

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Liste des utilisateurs</title>
  </head>
  <body>
    <h1>Liste des utilisateurs</h1>
    <table>
      <thead>
        <tr>
          <th>Nom</th>
          <th>Email</th>
          <th>Age</th>
        </tr>
      </thead>
      <tbody>
        <% users.forEach(function(user) { %>
          <tr>
            <td><%= user.name %></td>
            <td><%= user.email %></td>
            <td><%= user.age %></td>
          </tr>
        <% }); %>
      </tbody>
    </table>
    <form action="/users" method="post">
      <label>Nom : </label><input type="text" name="name"><br>
      <label>Email : </label><input type="email" name="email"><br>
      <label>Age : </label><input type="number" name="age"><br>
      <input type="submit" value="Ajouter un utilisateur">
    </form>
  </body>
</html>
```



# CRUD EXPRESS

---

Dans cette présentation, nous allons explorer la création d'un CRUD complet en utilisant Mongoose et Express. Nous allons couvrir les fonctionnalités Create, Read, Update et Delete.

# Create

---

```
const User = require('./models/user');

app.post('/users', async (req, res) => {
  const { name, type, size } = req.body;

  const user = new User({
    name,
    type,
    size
  });

  await user.save();

  res.status(201).json(user);
});
```

# Get

---

```
const User = require('./models/user');

app.get('/users', async (req, res) => {
  const users = await User.find();

  res.json(users);
});
```

# Update

---

```
const User = require('./models/user');

app.patch('/users/:id', async (req, res) => {
  const { id } = req.params;
  const { name, type, size } = req.body;

  const user = await User.findOneAndUpdate({ _id: id }, { name, type, size }, { new: true });

  res.json(user);
});
```

# Delete

---

```
const User = require('../models/user');

app.delete('/users/:id', async (req, res) => {
  const { id } = req.params;

  await User.findByIdAndDelete(id);

  res.sendStatus(204);
});
```



# Création d'un swagger

---

Dans cette présentation, nous allons explorer la création d'une documentation Swagger pour notre API RESTful créée avec Express. Swagger nous permet de documenter notre API de manière claire et structurée, et de fournir des exemples d'utilisation de nos endpoints.

# Installation

---

Pour commencer, nous devons installer Swagger dans notre projet.

Voici les étapes à suivre :

1. Installer la bibliothèque swagger-ui-express : `npm install swagger-ui-express`
2. Installer la bibliothèque swagger-jsdoc : `npm install swagger-jsdoc`

# Configuration

---

```
const swaggerUi = require('swagger-ui-express');
const swaggerJSDoc = require('swagger-jsdoc');

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Mon API RESTful',
      version: '1.0.0',
      description: 'Une description de mon API',
    },
    servers: [
      {
        url: 'http://localhost:3000',
      },
    ],
  },
  apis: ['./routes/*.js'],
};

const swaggerSpec = swaggerJSDoc(options);

app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerSpec));
```

# Définition des endpoints

---

```
✓ /**
 * @swagger
 * /users:
 *   get:
 *     summary: Récupère tous les utilisateurs
 *     responses:
 *       200:
 *         description: Liste des utilisateurs
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 type: object
 *                 properties:
 *                   name:
 *                     type: string
 *                   type:
 *                     type: string
 *                   size:
 *                     type: string
 *                 example:
 *                   name: "Le nom de la personne"
 *                   type: "belle"
 *                   size: "178cm"
 */
```

# Utilisation

---

Maintenant que nous avons documenté nos endpoints et nos modèles, nous pouvons accéder à l'interface Swagger en naviguant vers <http://localhost:3000/api-docs>.

L'interface nous permet de voir tous nos endpoints documentés, d'essayer les endpoints avec des exemples de requêtes et de réponses, et d'obtenir des informations supplémentaires sur notre API.



# Authentification

---

Dans cette présentation, nous allons explorer la mise en place d'une authentification avec token JWT (JSON Web Token) pour notre application Node.js avec Express. L'authentification avec JWT permet de sécuriser l'accès à notre application en générant un token d'accès qui peut être vérifié à chaque requête.

# Installation de dépendances

---

Pour commencer, nous devons installer les dépendances nécessaires pour utiliser JWT. Nous allons installer les bibliothèques jsonwebtoken et express-jwt. Voici les étapes à suivre :

1. Installer la bibliothèque jsonwebtoken : `npm install jsonwebtoken`
2. Installer la bibliothèque express-jwt : `npm install express-jwt`

# Le model

---

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true,
    },
    password: {
        type: String,
        required: true,
    },
});

const User = mongoose.model('User', userSchema);

module.exports = User;
```

# L'inscription

---

```
const bcrypt = require('bcrypt');
const User = require('./models/user');

app.post('/register', async (req, res) => {
  const { username, password } = req.body;

  const hashedPassword = await bcrypt.hash(password, 10);

  const user = new User({
    username,
    password: hashedPassword,
  });

  await user.save();

  res.status(201).send('Utilisateur créé avec succès');
});
```

# La connexion

---

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

const secret = 'mysecret';

app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = await User.findOne({ username });

  if (!user) {
    return res.status(401).send('Nom d\'utilisateur ou mot de passe incorrect');
  }

  const isPasswordCorrect = await bcrypt.compare(password, user.password);

  if (!isPasswordCorrect) {
    return res.status(401).send('Nom d\'utilisateur ou mot de passe incorrect');
  }

  const token = jwt.sign({ username }, secret, { expiresIn: '1h' });

  res.cookie('token', token, { httpOnly: true });

  res.status(200).send('Vous êtes connecté');
});
```

# Vérification du token JWT

---

```
app.get('/protected', jwtMiddleware, (req, res) => {
  const { username } = req.user;

  res.send(`Bienvenue, ${username}`);
});
```

# Customisation du middleware

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

const secret = 'mysecret';

function verifyToken(req, res, next) {
    const token = req.cookies.token;

    if (!token) {
        return res.status(401).send('Token manquant');
    }

    try {
        const decoded = jwt.verify(token, secret);
        req.user = decoded;
        next();
    } catch (err) {
        res.status(401).send('Token invalide');
    }
}

app.get('/protected', verifyToken, (req, res) => {
    const { username, role } = req.user;

    res.send(`Bienvenue, ${username} (${role})`);
});
```

# Bearer token

---

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

const secret = 'mysecret';

function verifyToken(req, res, next) {
    const authHeader = req.headers.authorization;
    const token = authHeader && authHeader.split(' ')[1];

    if (!token) {
        return res.status(401).send('Token manquant');
    }

    try {
        const decoded = jwt.verify(token, secret);
        req.user = decoded;
        next();
    } catch (err) {
        res.status(401).send('Token invalide');
    }
}

app.get('/protected', verifyToken, (req, res) => {
    const { username, role } = req.user;

    res.send(`Bienvenue, ${username} (${role})`);
});
```

# Documentation

---

```
const swaggerJSDoc = require('swagger-jsdoc');

const swaggerOptions = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'API',
      version: '1.0.0',
    },
    components: {
      securitySchemes: {
        bearerAuth: {
          type: 'http',
          scheme: 'bearer',
          bearerFormat: 'JWT',
        },
      },
    },
    apis: ['./routes/*.js'],
  };
};

const swaggerDocs = swaggerJSDoc(swaggerOptions);

app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocs));
```

# Documentation

---

```
/**
 * @swagger
 * /protected:
 *   get:
 *     security:
 *       - bearerAuth: []
 *     summary: Récupère les données protégées
 *     responses:
 *       200:
 *         description: Données récupérées avec succès
 */
app.get('/protected', verifyToken, (req, res) => {
  const { username, role } = req.user;

  res.send(`Bienvenue, ${username} (${role})`);
});
```



# Les sous collections

---

Les sous-collections sont une fonctionnalité de Mongoose qui permettent de stocker des données imbriquées dans un document parent. Dans cette présentation, nous allons explorer comment utiliser les sous-collections dans Mongoose.



# Définition du schéma

```
const userSchema = new mongoose.Schema({  
    name: String,  
    address: {  
        street: String,  
        city: String,  
        state: String,  
        zip: String,  
    },  
});
```



# Utilisation de la sous-collection

```
const User = mongoose.model('User', userSchema);

const user = new User({
  name: 'John Doe',
  address: {
    street: '123 Main St',
    city: 'Anytown',
    state: 'CA',
    zip: '12345',
  },
});

await user.save();
```



# La pagination

```
app.get('/users', async (req, res) => {
  const { page = 1, limit = 10 } = req.query;
  const skip = (page - 1) * limit;

  const users = await User.find().skip(skip).limit(parseInt(limit));

  res.json(users);
});
```

```
app.get('/users', async (req, res) => {
  const { page = 1, limit = 10 } = req.query;
  const skip = (page - 1) * limit;

  const users = await User.aggregate([
    { $skip: skip },
    { $limit: parseInt(limit) },
  ]);

  res.json(users);
});
```



# Communication en temps réel avec Socket.IO

---

Socket.IO est une bibliothèque JavaScript qui permet une communication en temps réel entre le client et le serveur. Elle est construite sur le protocole WebSocket, qui fournit un canal de communication bidirectionnel et persistant.

Pour utiliser Socket.IO dans une application, il est nécessaire de l'installer et de la configurer, puis de gérer les connexions et les déconnexions, d'émettre des événements et de diffuser des messages à d'autres utilisateurs connectés.



# Installation et configuration de Socket.IO (client/serveur)

Pour installer Socket.IO dans une application Node.js, il suffit d'exécuter la commande `npm install socket.io` dans le terminal. Une fois installée, il est nécessaire de configurer Socket.IO côté serveur en créant une instance de `io` et en l'écoutant sur un port. Voici un exemple de configuration Socket.IO côté serveur :

```
// Cr ation d'une instance de io
const io = require('socket.io')(server);

// Ecoute sur le port 3000
server.listen(3000, () => {
  console.log('Serveur d茅marr茅 sur le port 3000');
});
```



# Installation et configuration de Socket.IO (client/serveur)

Du côté client, il est également nécessaire d'inclure la bibliothèque Socket.IO dans la page HTML et de se connecter au serveur via une instance de io. Voici un exemple de configuration Socket.IO côté client :

```
<!-- Inclusion de la bibliothèque Socket.IO -->
<script src="/socket.io/socket.io.js"></script>
```

```
<!-- Connexion au serveur Socket.IO -->
<script>
  const socket = io();
</script>
```

```
// Connexion au serveur Socket.IO
const newSocket = io('http://localhost:3000');
```



# Installation et configuration de Socket.IO (client/serveur)

```
// Configuration de l'application Express
const express = require('express');
const app = express();

// Cr ation d'un serveur HTTP
const server = require('http').createServer(app);

// Cr ation d'une instance de Socket.IO pour la route /socket
const io = require('socket.io')(server, {
  path: '/socket'
});

// Gestion des connexions avec Socket.IO
io.on('connection', (socket) => {
  console.log(`Un utilisateur est connect  avec l'ID ${socket.id}`);

  // Gestion des d connexions
  socket.on('disconnect', () => {
    console.log(`L'utilisateur avec l'ID ${socket.id} s'est d connect `);
  });
});

// D marrage du serveur
server.listen(3000, () => {
  console.log('Serveur d marr  sur le port 3000');
});
```



# Gestion des connexions / déconnexions

Une fois que la connexion est établie entre le client et le serveur, il est possible de gérer les connexions et les déconnexions en utilisant les événements connect et disconnect. Ces événements permettent d'effectuer des actions lorsqu'un utilisateur se connecte ou se déconnecte. Voici un exemple de gestion de la connexion et de la déconnexion côté serveur :

```
// Gestion de la connexion
io.on('connect', (socket) => {
  console.log(`Utilisateur connecté avec l'ID ${socket.id}`);
});

// Gestion de la déconnexion
io.on('disconnect', (socket) => {
  console.log(`Utilisateur déconnecté avec l'ID ${socket.id}`);
});
```



# Emission d'événements

Socket.IO permet également d'émettre des événements personnalisés entre le client et le serveur. Pour émettre un événement, il suffit d'utiliser la méthode emit côté client et côté serveur. L'événement peut contenir des données qui peuvent être utilisées pour effectuer des actions spécifiques. Voici un exemple d'émission d'un événement côté client et côté serveur :

```
// Emission d'un événement côté client
socket.emit('mon_evenement', 'Données de l\'événement');

// Réception de l'événement côté serveur
io.on('connection', (socket) => {
  socket.on('mon_evenement', (data) => {
    console.log(`Données de l'événement reçues : ${data}`);
  });
});
```



# Broadcasting

---

Socket.IO offre la possibilité de diffuser des messages à tous les utilisateurs connectés via l'événement broadcast. Lorsqu'un événement est émis avec la méthode broadcast, il est envoyé à tous les utilisateurs connectés, à l'exception de l'utilisateur qui l'a émis. Voici un exemple d'utilisation de la méthode broadcast

```
// Emission d'un événement à tous les utilisateurs sauf à l'émetteur
socket.broadcast.emit('mon_evenement', 'Données de l\'événement');
```



# Multicasting avec les rooms

Enfin, Socket.IO permet la création de groupes d'utilisateurs appelés "rooms". Les utilisateurs peuvent être ajoutés à une room et les événements peuvent être émis et diffusés uniquement à ce groupe spécifique. Cela permet de limiter la diffusion de messages et de personnaliser l'expérience de chaque utilisateur. Voici un exemple d'utilisation des rooms :

```
// Création d'une room
socket.join('ma_room');

// Emission d'un événement à tous les utilisateurs dans la room
io.to('ma_room').emit('mon_evenement', 'Données de l\'événement dans la room')
```

# Configuration de l'environnement

---

- Installation d'Express.js et des dépendances nécessaires
- Configuration du serveur Express.js
- Création du répertoire d'upload

```
const express = require('express');
const app = express();
const port = 3000;

// Configuration du répertoire d'upload
app.use(express.static('uploads'));

app.listen(port, () => {
  console.log(`Serveur Express.js en cours d'exécution sur le port ${port}`)
});
```

# Gérer les requêtes d'upload

---

- Utilisation du middleware Multer pour gérer les requêtes d'upload

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('image'), (req, res) => {
  // Traitement de l'image uploadée
  console.log(req.file); // Informations sur l'image uploadée
  res.send('Image uploadée avec succès');
});
```

# Traitement des images uploadées

---

- Récupération des données de l'image uploadée
- Validation des types de fichiers et des tailles
- Génération d'un nom de fichier unique pour éviter les conflits
- Stockage des images sur le serveur ou utilisation d'un service de stockage cloud

```
// Exemple de validation du type de fichier (extension .jpg ou .png)
const supportedFileTypes = ['.jpg', '.png'];

app.post('/upload', upload.single('image'), (req, res) => {
  const file = req.file;
  const ext = path.basename(file.originalname).toLowerCase();

  if (!supportedFileTypes.includes(ext)) {
    return res.status(400).send('Le type de fichier n\'est pas pris en charge');
  }

  // Génération d'un nom de fichier unique
  const uniqueFileName = `${uuidv4()}${ext}`;

  // Stockage de l'image sur le serveur ou dans un service de stockage cloud

  res.send('Image uploadée avec succès');
});
```

# Affichage des images uploadées

---

- Récupération des images depuis le serveur ou le service de stockage cloud
- Affichage des images dans les vues ou les API
- Gestion de la mise en cache des images

```
app.get('/images/:filename', (req, res) => {
  const filename = req.params.filename;
  const imagePath = path.join(__dirname, 'uploads', filename);

  // Envoi de l'image en réponse
  res.sendFile(imagePath);
});
```

# Documentation

---

```
/**
 * @swagger
 * /upload:
 *   post:
 *     summary: Upload d'une image
 *     requestBody:
 *       description: Image à uploader
 *       required: true
 *       content:
 *         multipart/form-data:
 *           schema:
 *             type: object
 *             properties:
 *               image:
 *                 type: string
 *                 format: binary
 *     responses:
 *       200:
 *         description: Image uploadée avec succès
 */
```



# Optimisation des performances MongoDB

- Les index sont un moyen d'accélérer les requêtes de recherche dans une base de données MongoDB
- Mongoose prend en charge la création d'index pour les modèles de données
- Exemple de création d'index pour le champ "email" d'un modèle User:

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  username: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

userSchema.index({ email: 1 });

const User = mongoose.model('User', userSchema);

module.exports = User;
```



# Documents référencés

- Pour créer un modèle d'utilisateur avec une référence à un autre modèle, vous pouvez utiliser la méthode "Schema.Types.ObjectId" de Mongoose pour représenter l'identifiant unique d'un document dans une autre collection
- Exemple de modèle d'utilisateur avec une référence à un modèle de poste :

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  username: { type: String, required: true },
  posts: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Post' }]
});

const User = mongoose.model('User', userSchema);

module.exports = User;
```



# Création d'un modèle d'utilisateur

- Pour créer un modèle d'utilisateur avec une référence à un autre modèle, vous pouvez utiliser la méthode "Schema.Types.ObjectId" de Mongoose pour représenter l'identifiant unique d'un document dans une autre collection
- Exemple de modèle d'utilisateur avec une référence à un modèle de poste :

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  username: { type: String, required: true },
  posts: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Post' }]
});

const User = mongoose.model('User', userSchema);

module.exports = User;
```



# Création d'un modèle de poste

- Pour créer un modèle de poste avec une référence à un autre modèle, vous pouvez utiliser la méthode "Schema.Types.ObjectId" de Mongoose pour représenter l'identifiant unique d'un document dans une autre collection
- Exemple de modèle de poste avec une référence à un modèle d'utilisateur :

```
const mongoose = require('mongoose');

const postSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true }
});

const Post = mongoose.model('Post', postSchema);

module.exports = Post;
```



# Création de documents avec des références

- Pour créer un document avec une référence à un autre document, vous pouvez simplement stocker l'identifiant unique du document dans la propriété de référence du document
- Exemple de création d'un document de poste avec une référence à un document d'utilisateur :

```
const User = require('./models/user');
const Post = require('./models/post');

User.findOne({ username: 'johndoe' }, (err, user) => {
  if (err) {
    return console.error(err);
  }

  const post = new Post({
    title: 'My Post',
    content: 'Lorem ipsum dolor sit amet',
    author: user._id
  });

  post.save((err, post) => {
    if (err) {
      return console.error(err);
    }
    console.log(post);
  });
});
```



# Préchargement de documents référencés avec "populate"

- Pour précharger les documents référencés lors d'une requête, vous pouvez utiliser la méthode "populate" de Mongoose
- Exemple de requête pour récupérer un document de poste avec le document d'utilisateur référencé préchargé :

```
const Post = require('./models/post');

Post.findOne({ title: 'My Post' })
  .populate('author')
  .exec((err, post) => {
    if (err) {
      return console.error(err);
    }
    console.log(post);
});
```



# Méthode "lean"

- La méthode "lean" permet de récupérer des documents MongoDB en tant qu'objets JavaScript simples plutôt qu'en tant qu'instances de modèle Mongoose
- Cela peut améliorer les performances en réduisant la surcharge de l'objet Mongoose
- Exemple d'utilisation de la méthode "lean":

```
const Post = require('./models/post');

Post.find({})
  .lean()
  .exec((err, posts) => {
    if (err) {
      return console.error(err);
    }
    console.log(posts);
});
```



# Méthode "select"

- La méthode "select" permet de limiter les champs renvoyés par une requête
- Cela peut améliorer les performances en réduisant la quantité de données transférées sur le réseau
- Exemple d'utilisation de la méthode "select":

```
const Post = require('./models/post');

// Renvoie tous les documents de la collection "posts" mais en excluant le champ "content"
Post.find({}, { content: 0 }, (err, posts) => {
  if (err) {
    return console.error(err);
  }
  console.log(posts);
});
```



# Qu'est-ce que Joi ?

- Joi est une bibliothèque de validation de schémas pour JavaScript
- Elle est utilisée pour valider les données entrantes dans les requêtes HTTP dans Express
- Elle fournit des schémas prédéfinis pour les types de données courants tels que les chaînes, les nombres et les tableaux, ainsi que la possibilité de créer des schémas personnalisés pour des types de données plus complexes.



# Utilisation de Joi dans Express

Pour utiliser Joi dans Express, vous devez d'abord importer la bibliothèque :

```
const Joi = require('joi');
```

Ensuite, vous pouvez créer un schéma de validation pour valider les données entrantes dans une requête :

```
const schema = Joi.object({
  username: Joi.string()
    .alphanum()
    .min(3)
    .max(30)
    .required(),
  password: Joi.string()
    .pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')),
  repeat_password: Joi.ref('password'),
  access_token: [
    Joi.string(),
    Joi.number()
  ],
  birth_year: Joi.number()
    .integer()
    .min(1900)
    .max(2013),
  email: Joi.string()
    .email({ minDomainSegments: 2, tlds: { allow: ['com', 'net'] } })
})
```



# Utilisation de Joi dans Express

Pour utiliser ce schéma dans une route Express, nous pouvons simplement ajouter une fonction middleware de validation qui utilise le schéma de validation :

```
app.post('/signup', (req, res, next) => {
  const { error, value } = signupSchema.validate(req.body);
  if (error) {
    res.status(422).json({ message: error.details[0].message });
  } else {
    // Les données sont valides, on peut les utiliser pour créer un nouvel utilisateur
    // ...
  }
});
```



# Middleware express

```
const Joi = require('joi');

// Factory pour créer des middlewares de validation de schéma
function schemaValidator(schema) {
  return (req, res, next) => {
    const { error, value } = schema.validate(req.body);
    if (error) {
      res.status(422).json({ message: error.details[0].message });
    } else {
      // Les données sont valides, on passe au middleware suivant
      req.body = value; // On remplace le corps de la requête par les données
      next();
    }
  };
}

// Définition des schémas de validation pour chaque endpoint
const signupSchema = Joi.object({
  username: Joi.string().alphanumeric().min(3).max(30).required(),
  password: Joi.string().pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')).required(),
  email: Joi.string().email({ minDomainSegments: 2, tlds: { allow: ['com', 'fr'] } })
});

const loginSchema = Joi.object({
  username: Joi.string().alphanumeric().min(3).max(30).required(),
  password: Joi.string().pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')).required()
});

// Routes avec validation de schéma différente pour chaque endpoint
app.post('/signup', schemaValidator(signupSchema), (req, res) => {
  // Les données ont été validées, on peut créer un nouvel utilisateur
});

app.post('/login', schemaValidator(loginSchema), (req, res) => {
  // Les données ont été validées, on peut authentifier l'utilisateur
});
```



# Pourquoi utiliser Mocha pour écrire des tests unitaires ?

- Mocha facilite l'écriture de tests unitaires en fournissant une syntaxe simple et concise
  - Il est hautement configurable, permettant aux développeurs de personnaliser les tests en fonction de leurs besoins
  - Il dispose d'un grand écosystème de plugins et d'outils d'intégration avec d'autres bibliothèques populaires, tels que Chai et Sinon.
- 
- `npm install --save-dev mocha`



# Exemple d'utilisation de Mocha

Supposons que nous avons une fonction addNumbers qui prend deux nombres en entrée et les ajoute. Nous pouvons utiliser Mocha pour écrire des tests unitaires pour cette fonction. Voici un exemple de test unitaire avec Mocha pour la fonction addNumbers :

```
const assert = require('assert');

describe('Fonction addNumbers', function() {
  it('devrait retourner la somme de deux nombres', function() {
    const a = 2;
    const b = 3;
    const result = addNumbers(a, b);

    assert.equal(result, 5);
  });
});
```



## Les méthodes disponibles dans l'objet assert

- `assert.equal(actual, expected[, message])` : vérifie que actual est égal à expected
- `assert.strictEqual(actual, expected[, message])` : vérifie que actual est strictement égal à expected
- `assert.notEqual(actual, expected[, message])` : vérifie que actual n'est pas égal à expected
- `assert.notStrictEqual(actual, expected[, message])` : vérifie que actual n'est pas strictement égal à expected
- `assert.ok(value[, message])` : vérifie que value est vrai (ou équivalent à vrai)
- `assert.fail([message])` : fait échouer le test avec un message d'erreur personnalisé



## Quelques méthodes disponibles dans l'objet assert

- `assert.equal(actual, expected[, message])` : vérifie que actual est égal à expected
- `assert.strictEqual(actual, expected[, message])` : vérifie que actual est strictement égal à expected
- `assert.notEqual(actual, expected[, message])` : vérifie que actual n'est pas égal à expected
- `assert.notStrictEqual(actual, expected[, message])` : vérifie que actual n'est pas strictement égal à expected
- `assert.ok(value[, message])` : vérifie que value est vrai (ou équivalent à vrai)
- `assert.fail([message])` : fait échouer le test avec un message d'erreur personnalisé



# Utilisation de before et after pour exécuter du code avant et après les tests

- before et after permettent d'exécuter du code avant et après l'exécution de tous les tests dans une suite de tests
- Ils peuvent être utilisés pour mettre en place et nettoyer les ressources nécessaires aux tests
- Voici un exemple d'utilisation de before et after pour créer et détruire une connexion de base de données avant et après l'exécution de tous les tests :

```
const assert = require('assert');
const mongoose = require('mongoose');

before(function() {
  mongoose.connect('mongodb://localhost/test');
});

after(function() {
  mongoose.connection.close();
});

describe('Fonction de test', function() {
  it('devrait faire quelque chose', function() {
    // Test
  });
});
});
```



## Utilisation de beforeEach et afterEach pour exécuter du code avant et après chaque cas de test

- beforeEach et afterEach permettent d'exécuter du code avant et après chaque cas de test dans une suite de tests
- Ils peuvent être utilisés pour initialiser ou réinitialiser des variables et des objets entre chaque test
- Voici un exemple d'utilisation de beforeEach et afterEach pour réinitialiser une variable avant et après chaque test :

```
const assert = require('assert');
let count = 0;

beforeEach(function() {
  count++;
});

afterEach(function() {
  count = 0;
});

describe('Fonction de test', function() {
  it('devrait augmenter la valeur de count', function() {
    assert.equal(count, 1);
  });

  it('devrait réinitialiser la valeur de count', function() {
    assert.equal(count, 1);
  });
});
```



# Utilisation de skip pour ignorer des tests

- skip permet d'ignorer un ou plusieurs tests dans une suite de tests
- Il peut être utilisé pour désactiver temporairement des tests qui ne fonctionnent pas ou qui ne sont pas encore implémentés
- Voici un exemple d'utilisation de skip pour ignorer temporairement un test :

```
const assert = require('assert');

describe('Fonction de test', function() {
  it.skip('devrait être implémenté prochainement', function() {
    // Test non implémenté
  });

  it('devrait faire quelque chose', function() {
    assert.ok(true);
  });
});
```



# Sequelize : ORM pour Node.js

---

- Sequelize est un ORM (Object-Relational Mapping) pour Node.js.
- Il facilite l'interaction avec les bases de données relationnelles en utilisant des objets JavaScript.
- Il prend en charge plusieurs types de bases de données, notamment PostgreSQL, MySQL, MariaDB et SQLite.

```
const Sequelize = require('sequelize');

const sequelize = new Sequelize('postgres://localhost:5432/mydb');

// Définir un modèle
const User = sequelize.define('user', {
  username: Sequelize.STRING,
  email: Sequelize.STRING,
  password: Sequelize.STRING
});

// Synchroniser le modèle avec la base de données
User.sync();
```



# Fonctionnalités principales de Sequelize

- Définition de modèles
- Déclaration des champs
- Définition des relations entre les modèles
- Validation des données
- Opérations CRUD (Create, Read, Update, Delete)
- Chargement des relations
- Utilisation des relations dans les requêtes
- Fonctions avancées : transactions, hooks, requêtes brutes

```
// Créer un utilisateur
const user = await User.create({
  username: 'johndoe',
  email: 'johndoe@example.com',
  password: 'password123'
});

// Lire un utilisateur
const user = await User.findByPk(1);

// Mettre à jour un utilisateur
user.username = 'janedoe';
await user.save();

// Supprimer un utilisateur
await user.destroy();
```



# Comparaison avec d'autres ORMs

- Sequelize est un ORM populaire, mais il existe d'autres options disponibles.
- Voici quelques points de comparaison entre Sequelize et d'autres ORMs :

ORM	Fonctionnalités	Facilité d'utilisation	Performances	Documentation
Sequelize	Large gamme de fonctionnalités	Bonne	Bonnes	Bonne
Mongoose	Orienté NoSQL	Bonne	Excellent	Bonne
Bookshelf	Léger et simple	Bonne	Bonnes	Moyenne



# Installation et configuration de Sequelize

---

- Installer Sequelize avec `npm install sequelize`.
- Créer une instance Sequelize et configurer la connexion à la base de données.
- Définir les modèles en utilisant des classes JavaScript.



# Définition des modèles avec Sequelize

- Les modèles Sequelize représentent des tables dans votre base de données.
- Chaque champ dans un modèle correspond à une colonne dans la table correspondante.
- Sequelize propose différents types de données pour définir les champs de vos modèles, tels que STRING, INTEGER, BOOLEAN, etc.



# Déclaration des champs dans un modèle Sequelize

---

- Utilisez la méthode Sequelize.STRING pour les chaînes de texte.
- Utilisez la méthode Sequelize.INTEGER pour les nombres entiers.
- Utilisez la méthode Sequelize.BOOLEAN pour les valeurs booléennes (true/false).
- D'autres types de données sont disponibles pour différents besoins.

```
const Sequelize = require('sequelize');

const User = sequelize.define('user', {
  username: Sequelize.STRING,
  email: Sequelize.STRING,
  age: Sequelize.INTEGER,
  isAdmin: Sequelize.BOOLEAN
});
```



# Définition des relations entre les modèles Sequelize

---

- Permet de modéliser les relations entre différentes entités dans votre base de données.
- Relation "un-à-plusieurs" (One-to-Many): Un modèle peut être associé à plusieurs instances d'un autre modèle.
- Relation "plusieurs-à-plusieurs" (Many-to-Many): Plusieurs instances d'un modèle peuvent être associées à plusieurs instances d'un autre modèle.

```
const User = sequelize.define('user', {
  // ...
});

const Post = sequelize.define('post', {
  // ...
});

// Relation un-à-plusieurs: un utilisateur peut avoir plusieurs posts
User.hasMany(Post);
Post.belongsTo(User);

// Relation plusieurs-à-plusieurs: un utilisateur peut avoir plusieurs tags
const UserTag = sequelize.define('user_tag', {});

User.belongsToMany(Tag, { through: UserTag });
Tag.belongsToMany(User, { through: UserTag });
```



# Validation des données avec Sequelize

- Garantit l'intégrité et la cohérence des données dans votre base de données.
- Sequelize propose des validations intégrées pour les champs de modèle.
- Exemples de validations :
- Obligatoire (required): Spécifie qu'un champ ne peut pas être vide.
- Longueur minimale/maximale (min/max): Définit la longueur minimale ou maximale autorisée pour une chaîne de texte.
- Email : Vérifie qu'une chaîne de texte est une adresse email valide.

```
const User = sequelize.define('user', {
  username: {
    type: Sequelize.STRING,
    allowNull: false, // obligatoire
    unique: true // unique
  },
  email: {
    type: Sequelize.STRING,
    allowNull: false,
    validate: {
      isEmail: true // email valide
    }
  },
  age: {
    type: Sequelize.INTEGER,
    min: 18 // âge minimum
  }
});
```



# Opérations CRUD

---

- Create (Créer): Insérer de nouvelles entités dans la base de données.
- Read (Lire): Récupérer des entités existantes de la base de données.
- Update (Mettre à jour): Modifier les données d'entités existantes.
- Delete (Supprimer): Supprimer des entités de la base de données.

```
// Créer un utilisateur
const user = await User.create({
  username: 'johndoe',
  email: 'johndoe@example.com',
  age: 30
});

// Lire un utilisateur
const user = await User.findByPk(1);

// Mettre à jour un utilisateur
user.username = 'janedoe';
await user.save();

// Supprimer un utilisateur
await user.destroy();
```



# Relations entre les modèles

- Chargement des relations: Récupérer les données associées d'un modèle en même temps que l'entité principale.
- Utilisation des relations dans les requêtes: Effectuer des requêtes en tenant compte des relations entre les modèles.

```
// Charger les posts d'un utilisateur
const user = await User.findByPk(1, {
  include: [Post] // inclut les posts associés
});

console.log(user.posts); // liste des posts de l'utilisateur

// Requête pour récupérer les utilisateurs avec leurs posts récents
const users = await User.findAll({
  include: [
    {
      model: Post,
      where: {
        createdAt: {
          [Op.gt]: Date.now() - 7 * 24 * 60 * 60 * 1000 // il y a 7 jours
        }
      }
    }
  ]
});
```



# Fonctions avancées de Sequelize

---

- Transactions: Permettent d'exécuter plusieurs opérations de base de données de manière atomique, garantissant ainsi la cohérence des données.
- Hooks: Fonctions qui s'exécutent automatiquement à certains moments du cycle de vie d'un modèle, permettant d'exécuter des actions avant ou après certaines opérations CRUD.
- Requêtes brutes: Permettent d'exécuter des requêtes SQL directement sur la base de données, offrant une flexibilité supplémentaire pour des cas d'utilisation spécifiques.



# Transactions:

```
const sequelize = await sequelize.transaction(async (t) => {
  // Opérations CRUD dans la transaction
  await User.create({ username: 'johndoe' }, { transaction: t });
  await Post.create({ title: 'My first post' }, { transaction: t });

  // Si une erreur survient, la transaction est annulée
  // Sinon, la transaction est validée et les modifications sont commises
});
```



# Hooks :



```
User.beforeCreate((user) => {
  // Fonction exécutée avant la création d'un utilisateur
  user.username = user.username.toLowerCase();
});
```



## Requêtes brutes :

```
const results = await sequelize.query('SELECT * FROM users WHERE age > 30')
```



# Requêtes de base

- findAll: récupération de tous les enregistrements d'un modèle
- findOne: récupération d'un seul enregistrement par son identifiant
- findByPk: récupération d'un enregistrement par sa clé primaire
- create: création d'un nouvel enregistrement
- update: mise à jour d'un enregistrement existant
- destroy: suppression d'un enregistrement

```
// findAll
const users = await User.findAll();

// findOne
const user = await User.findOne({ where: { id: 1 } });

// findByPk
const user = await User.findByPk(1);

// create
const user = await User.create({ name: 'John Doe', email: 'johndoe@example.com' });

// update
await user.update({ name: 'Jane Doe' });

// destroy
await user.destroy();
```



# Requêtes de base

- `findAndCountAll`: récupération de tous les enregistrements et du nombre total
- `findAllByPk`: récupération de plusieurs enregistrements par leurs clés primaires
- `findOrCreate`: recherche d'un enregistrement et création s'il n'existe pas
- `build`: création d'une instance de modèle sans la sauvegarder
- `save`: sauvegarde d'une instance de modèle
- `reload`: rechargement des données d'un enregistrement depuis la base de données

```
// findAndCountAll
const { rows, count } = await User.findAndCountAll();

// findAllByPk
const users = await User.findAllByPk([1, 2, 3]);

// findOrCreate
const [user, created] = await User.findOrCreate({ where: { email: 'john.doe@example.com' } });

// build
const user = User.build({ name: 'John Doe', email: 'johndoe@example.com' });

// save
await user.save();

// reload
await user.reload();
```



# Opérateurs de comparaison

- `=:` égalité
- `!=:` différence
- `>:` supérieur à
- `<:` inférieur à
- `>=:` supérieur ou égal à
- `<=:` inférieur ou égal à

```
const users = await User.findAll({
  where: {
    age: {
      // Supérieur à 18 ans
      gt: 18,
    },
    email: {
      // Ne commence pas par "admin"
      notLike: 'admin%',
    },
  },
});
```



# Authentification JWT avec Sequelize

---

- JWT est une solution populaire pour l'authentification des API car il est léger, flexible et sécurisé.
- Sequelize est un ORM (Object-Relational Mapping) pour Node.js qui facilite l'interaction avec les bases de données relationnelles.
- Cette combinaison permet de créer des systèmes d'authentification robustes et évolutifs.



# Fonctionnement l'authentification JWT

---

- Un token JWT est une chaîne de caractères encodé qui contient des informations sur l'utilisateur et l'émetteur du token.
- Il est composé de trois parties séparées par des points :
- En-tête: Contient des informations sur le type de token et l'algorithme de signature utilisé.
- Payload: Contient les données de l'utilisateur, telles que son identifiant, son nom et son rôle.
- Signature: Permet de vérifier l'intégrité du token et de s'assurer qu'il n'a pas été modifié.



# Avantages et inconvénients de l'authentification JWT

---

- Avantages:
- Léger et performant
- Sécurisé grâce à la signature
- Flexible et adaptable à différents cas d'utilisation
- Ne nécessite pas de stockage côté serveur des sessions utilisateur
- Inconvénients:
- La clé secrète doit être gardée confidentielle
- Les tokens peuvent être révoqués difficilement
- La taille du token peut être un problème pour certaines applications



# Création des modèles User et Token

- Définir les modèles User et Token avec Sequelize.
- Le modèle User doit contenir les informations de l'utilisateur, telles que son identifiant, son nom, son email et son mot de passe.
- Le modèle Token doit contenir le token JWT et l'identifiant de l'utilisateur associé.

```
const User = sequelize.define('user', {  
    username: Sequelize.STRING,  
    email: Sequelize.STRING,  
    password: Sequelize.STRING  
});
```

```
const Token = sequelize.define('token', {  
    token: Sequelize.STRING,  
    userId: Sequelize.INTEGER  
});
```

```
// Relation entre les modèles  
UserhasMany(Token);  
Token.belongsTo(User);
```



# Génération du token JWT

- Utiliser une bibliothèque comme jsonwebtoken pour générer le token.
- Le token doit contenir les informations de l'utilisateur et être signé avec une clé secrète.

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

const token = jwt.sign({
  id: user.id,
  username: user.username
}, secretKey, {
  expiresIn: '1h' // expiration du token après 1 heure
});
```



# Stockage du token dans la base de données

---

- Stockage:
- Stocker le token JWT dans la base de données pour éviter son envoi dans chaque requête.
- Associer le token à l'utilisateur correspondant.
- Options de stockage:
- Colonne dédiée dans la table User.
- Table séparée Token.



# Authentification des requêtes

- Inclure le token JWT dans l'en-tête de la requête (généralement dans un champ nommé "Authorization").
- Extraire le token de l'en-tête.
- Vérifier la validité du token en utilisant la clé secrète et la bibliothèque jsonwebtoken.
- Si le token est valide, récupérer les informations de l'utilisateur à partir du payload.
- Si le token est invalide, renvoyer une erreur d'authentification.

```
const auth = async (req, res, next) => {
  const token = req.headers['authorization'];
  if (!token) {
    return res.status(401).json({ message: 'Unauthorized' });
  }

  try {
    const decoded = jwt.verify(token, secretKey);
    req.user = decoded;
    next();
  } catch (err) {
    return res.status(401).json({ message: 'Invalid token' });
  }
};
```



# Sécurisation de l'API

---

- Middleware d'authentification:
- Utiliser un middleware pour vérifier le token JWT sur chaque requête protégée.
- Le middleware doit être placé avant les routes nécessitant une authentification.
- Gestion des erreurs d'authentification:
- Renvoyer des codes d'erreur HTTP appropriés (401 - Unauthorized, 403 - Forbidden).
- Fournir des messages d'erreur clairs et concis à l'utilisateur.



# Gestion des mots de passe

- Les mots de passe constituent la première ligne de défense contre les accès non autorisés.
- De nombreuses cyberattaques exploitent des mots de passe faibles ou compromis.
- Il est essentiel de mettre en place des mesures de sécurité adéquates pour protéger les mots de passe et les données des utilisateurs.



# Stockage sécurisé des mots de passe

- Ne jamais stocker les mots de passe en texte clair.
- Utiliser des techniques de hachage et de salage pour obscurcir les mots de passe stockés.
- Le hachage est une fonction à sens unique qui convertit un mot de passe en une valeur
- Le salage ajoute une valeur aléatoire au mot de passe avant le hachage, pour renforcer la sécurité.

```
const bcrypt = require('bcrypt');

const hashPassword = async (password) => {
  const salt = await bcrypt.genSalt(10);
  const hashedPassword = await bcrypt.hash(password, salt);
  return hashedPassword;
};

const comparePassword = async (password, hashedPassword) => {
  return await bcrypt.compare(password, hashedPassword);
};
```



# Réinitialisation des mots de passe

---

- Processus permettant aux utilisateurs de récupérer l'accès à leur compte en cas de mot de passe oublié.
- Étapes du processus de réinitialisation:
  - L'utilisateur saisit son adresse email ou son nom d'utilisateur.
  - Un jeton de réinitialisation est généré et envoyé à l'utilisateur.
  - L'utilisateur clique sur le lien dans le jeton et définit un nouveau mot de passe.
- Sécurisation du processus de réinitialisation:
  - Expiration des jetons: Définir une durée d'expiration pour les jetons de réinitialisation.
  - Jetons uniques: Générer des jetons uniques et aléatoires pour chaque utilisateur.
  - HTTPS: Utiliser le protocole HTTPS pour sécuriser la transmission des données.



# Réinitialisation des mots de passe

---

```
const nodemailer = require('nodemailer');

const sendResetEmail = async (email, token) => {
  const transporter = nodemailer.createTransport({
    // ... configuration du transport SMTP
  });

  const mailOptions = {
    from: 'noreply@example.com',
    to: email,
    subject: 'Réinitialisation de votre mot de passe',
    text: `Cliquez sur ce lien pour réinitialiser votre mot de passe
${process.env.RESET_URL}/reset?token=${token}`
  };

  await transporter.sendMail(mailOptions);
};
```



# Réinitialisation des mots de passe

---

- Processus permettant aux utilisateurs de récupérer l'accès à leur compte en cas de mot de passe oublié.
- Étapes du processus de réinitialisation:
  - L'utilisateur saisit son adresse email ou son nom d'utilisateur.
  - Un jeton de réinitialisation est généré et envoyé à l'utilisateur.
  - L'utilisateur clique sur le lien dans le jeton et définit un nouveau mot de passe.
- Sécurisation du processus de réinitialisation:
  - Expiration des jetons: Définir une durée d'expiration pour les jetons de réinitialisation.
  - Jetons uniques: Générer des jetons uniques et aléatoires pour chaque utilisateur.
  - HTTPS: Utiliser le protocole HTTPS pour sécuriser la transmission des données.



# Traitement de la connexion

- Récupération des informations d'identification: Extrait l'email et le mot de passe de la requête HTTP.
- Validation des informations d'identification: Recherche l'utilisateur dans la base de données et compare le mot de passe haché.
- Génération du token JWT: Si l'utilisateur est authentifié, génère un token JWT avec les informations de l'utilisateur.
- Envoi du token: Envoie le token JWT à l'utilisateur dans le corps de la réponse.

```
app.post('/login', async (req, res) => {
  const { email, password } = req.body;

  const user = await User.findOne({ where: { email } });

  if (!user || !bcrypt.compareSync(password, user.password)) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }

  const token = jwt.sign({ id: user.id }, secretKey, { expiresIn: '1h' });
  res.json({ token });
});
```



# Migrations de base de données avec Sequelize

---

- Les migrations de base de données permettent de gérer l'évolution du schéma de votre base de données au fil du temps.
- Sequelize offre un outil de migration intégré pour simplifier ce processus.
- Les migrations présentent de nombreux avantages, tels que la possibilité de suivre les modifications, de les annuler et de les déployer en production.

```
// Sequelize CLI  
sequelize migration:create --name add-user-table
```



# Créer et exécuter des migrations

- Les fichiers de migration sont des fichiers JavaScript qui exportent deux fonctions: up et down.
- La fonction up définit les modifications à apporter à la base de données.
- La fonction down définit comment annuler ces modifications.
- Vous pouvez utiliser l'interface de ligne de commande Sequelize pour créer, exécuter et annuler des migrations.

```
// Fichier de migration
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Users', {
      id: {
        type: Sequelize.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },
      name: {
        type: Sequelize.STRING,
        allowNull: false,
      },
      email: {
        type: Sequelize.STRING,
        unique: true,
        allowNull: false,
      },
    });
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Users');
  },
};
```



# Gérer les migrations

- Sequelize utilise une table interne SequelizeMeta pour stocker les noms des migrations appliquées.
- Vous pouvez utiliser la méthode getMigrations() pour obtenir une liste des migrations appliquées.
- Vous pouvez comparer cette liste avec la liste de toutes les migrations disponibles pour identifier les migrations non appliquées.

```
// Suivi des migrations appliquées
const sequelize = new Sequelize(...);
const migrations = await sequelize.getMigrations();

// Consulter les migrations appliquées
console.log(migrations.map(m => m.name));

// Identifier les migrations non appliquées
const allMigrations = await sequelize.query('SELECT name FROM SequelizeMeta');
const appliedMigrations = migrations.map(m => m.name);
const nonAppliedMigrations = allMigrations.filter(m => !appliedMigration.includes(m.name));

console.log(nonAppliedMigrations);
```