

Node.js

Node.js et l'accès aux données



plb consultant

SQL vs NOSQL

- SQL organise le stockage de données sur le principe de tables reliées entre elles.
- noSQL stocke et manipule des documents qui correspondent à des collections d'objets.

SQL vs NOSQL

Les tables SQL imposent un modèle de données strictes, donc il est difficile de faire des erreurs.

NoSQL est plus flexible et pardonnable, mais la possibilité de stocker des données n'importe où peut entraîner des problèmes de cohérence.

Les avantages d'une base de données NoSQL

- Flexibilité
- Évolutivité
- De hautes performances
- Disponibilité
- Hautement fonctionnel

Les types de bases de données NoSQL

- Valeur clé
- Document
- Graphe
- Colonne large

Plan

1. MongoDB
2. Mongoose
3. Waterline
4. Node.js avec Typescript
5. Tests unitaires avec Jest
6. Authentification avec Passport.js

MongoDb

La base de données

- multiplateforme orientée documents.
- travaille sur le concept de collection et de document.
- un conteneur physique pour les collections.

MongoDb

La collection :

- un groupe de documents MongoDB.
- Les documents d'une collection peuvent avoir différents champs.
- En règle générale, tous les documents d'une collection ont un objectif similaire ou connexe.
- Un document est un ensemble de paires clé-valeur.

MongoDb

Un document :

- Un ensemble de paires clé-valeur.
- Un schéma dynamique.

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100,
  comments: [
    {
      user: 'user1',
      message: 'My first comment',
      dateCreated: new Date(2011,1,20,2,15),
      like: 0
    },
    {
      user: 'user2',
      message: 'My second comments',
      dateCreated: new Date(2011,1,25,7,45),
      like: 5
    }
  ]
}
```

MongoDb

Télécharger : <https://www.mongodb.com/try/download/community>

Vérifier l'installation : C:\Program Files\MongoDB\Server\6.0\bin\mongo.exe

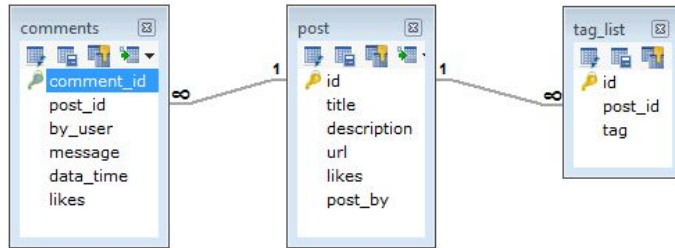
MongoDb

Exemple : Le site Web a les exigences suivantes.

- Chaque article à un titre, une description et une URL uniques.
- Chaque message peut avoir un ou plusieurs tags.
- Chaque message a le nom de son éditeur et le nombre total de likes.
- Chaque publication contient des commentaires donnés par les utilisateurs avec leur nom, leur message, leur heure de données et leurs goûts.
- Sur chaque message, il peut y avoir zéro ou plusieurs commentaires.

MongoDb

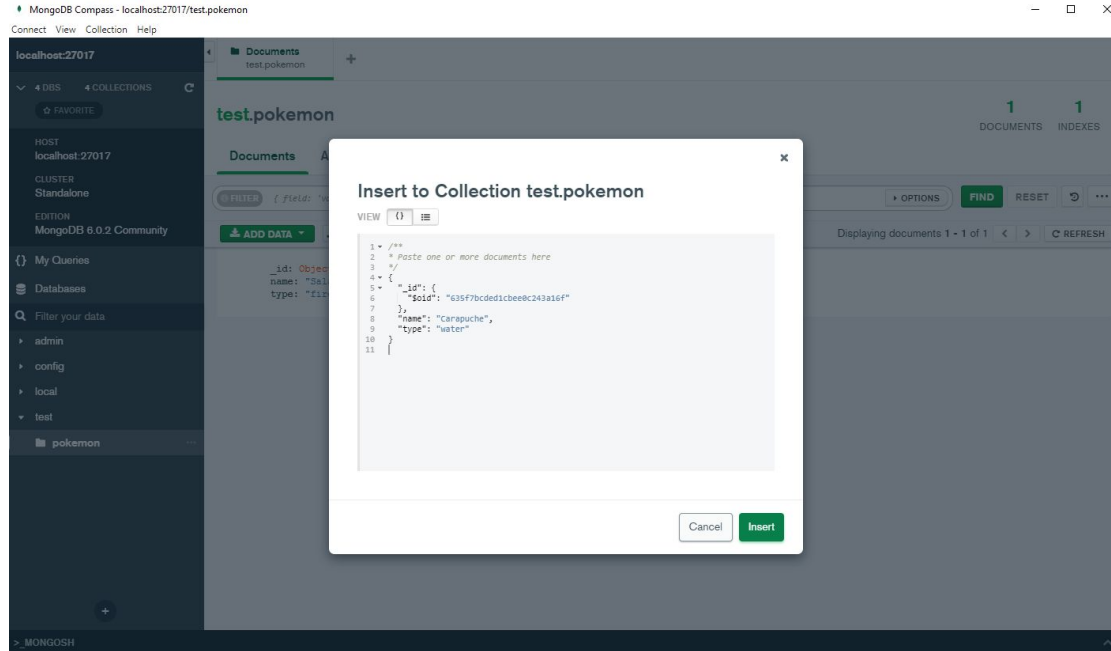
mySql :



MongoDb :

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  comments: [
    {
      user: 'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    },
    {
      user: 'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```

MongoDb



Mongoose

Installation : npm install mongoose --save

```
import mongoose from 'mongoose';

class Database {
  mongoDb = 'mongodb://127.0.0.1/test';

  connect() {
    mongoose.connect('mongodb://127.0.0.1:27017/test');
    // Get the default connection
    const db = mongoose.connection;

    // Bind connection to error event (to get notification of connection errors)
    db.on("error", console.error.bind(console, "MongoDB connection error:"));
  }
}

export default new Database();
```

Mongoose

```
import { Schema, model } from 'mongoose';

const pokemonSchema = new Schema({
  name: {
    type: String,
    required: [true, "Why no name?"],
  },
  type: {
    type: String,
    required: [true, "Why no type?"],
  }
});

const Pokemon = model('pokemon', pokemonSchema);
export default Pokemon;
```

Mongoose

```
import express from 'express';  
import Database from './config/Database.js';  
import Pokemon from './Model/Pokemon.js';
```

```
Database.connect();
```

```
const pokemon = new Pokemon({  
  name: 'Salameche',  
  type: 'Fire'  
})
```

```
await pokemon.save()
```

```
const app = express();  
//setting view engine to ejs  
app.set("view engine", "ejs");
```

```
app.get("/pokemon", async function (req, res) {  
  const filter = {};  
  let pokemons = await Pokemon.find(filter);  
  console.log(pokemons);  
  const titre = 'Mon Pokédex';  
  
  res.render("Pokemon", {  
    pokemons: pokemons,  
    title: titre  
  });  
});
```


Mongoose

Requête :

- `objet.save()`
- `objet.find()`
- `const query = Character.find({ name: 'Jean-Luc Picard' });`
- `query.getFilter();`
- `Auth.findOne({nick: 'noname'}, function(err,obj) { console.log(obj); });`

Exercice

- Afficher les films sur une page
- Ajouter une entité à la BDD
- Modifier une entité de la BDD
- Supprimer une entité de la BDD
- Ajouter un champ qui permet de filtrer les films par année
- Ajouter un champ qui permet de filtrer les films par note

Exercise

[Movies Dataset](#) [Home](#) [Configuration](#)

Select a Query



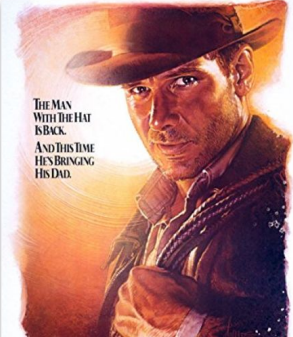
Query on title (select all the movies that contains your word(s) in the title) ▼

Title

Indiana

Search

4 movies found with your query



Waterline

<https://sailsjs.com/documentation/concepts/models-and-orm>

- Mappeur de données SQL/noSQL (ORM/ODM)

Typescript

- JavaScript est un langage complet et flexible:
- Son typage dynamique a ses forces mais peut rendre le code moins lisible et entraîner des erreurs.
- Il peut devenir compliqué de structurer le code dans les gros projets.
- JavaScript n'est pas orienté objet :Les classes sont des prototypes. Tout est public
- TypeScript permet d'ajouter des notions de POO et résoudre ces problèmes.



Typescript

Typescript fournit :

- Un typage fort : Permet d'éviter les erreurs de programmation
- La visibilité sur les attributs et méthodes des classes : Permet une meilleure structure de code
- De vraies classes : Proche du Java ou du C#
- Le polymorphisme : Interfaces, Classes abstraites, Type Générique
- Et garde les fonctionnalités d'ES6

Typescript

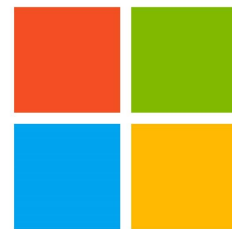
Typescript fournit :

La Programmation Orientée Objet permet de:

- Mieux structurer le code
- Rendre le code plus robuste
- Mettre en place des design-patterns

TypeScript est développé et maintenu par Microsoft: Il est sous License Apache 2.0 donc:

- Utilisable commercialement
- Modifiable
- Distribuible



Microsoft

TypeScript

Niveau performances:

- TypeScript est comparable aux performances de NodeJS
 - Il peut même être plus rapide sur certaines tâches grâce aux optimisations de l'orienté objet.
- <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/typescript.html>
- De nombreux frameworks utilisent TypeScript comme Angular et NestJS



Typescript

TypeScript utilise des fichiers '.ts' :

- Ils ressemblent fortement aux '.js'
- La syntaxe reste proche du JavaScript mais la forme des classes se rapproche du Java
- Ils utilisent un typage pour les variables. (typage à droite contrairement aux langages java ou C avec un typage à gauche)
- L'écriture des fonctions reste très proche du javascript :
 - Il est toujours possible de déclarer des fonctions fléchées
 - Le mot clef 'function' est toujours déclaré
 - L'asynchrone de nodeJS reste disponible.
- TypeScript doit être transpilé en JavaScript
- Il est possible d'utiliser du code Javascript dans du TypeScript.

TypeScript

- Pour utiliser typescript dans un projet il est nécessaire de l'installer avec npm:

```
npm install -g typescript
```

- TypeScript vient avec un CLI pour l'exécution du code:

- Pour lancer le module 'hello.ts' :

```
npx tsc hello.ts
```

- Il est aussi possible, voire préférable d'utiliser webpack pour la transpilation:

- Voir le support Webpack



Typescript

- TypeScript étant typé, pour l'utilisation de modules javascript il est souvent nécessaire d'installer en plus une dépendance contenant les types.
- Les dépendances de types ne sont pas nécessaires pour le build du projet. Par conséquent, on les installe en tant que devDependencies.
 - Pour les types de base de node: `npm install --save-dev @types/node`
 - Pour les types d'express : `npm install --save-dev @types/express`
 - En règle générale pour un module XXX l'on doit importer `@type/XXX`



webpack

TypeScript

Les variables :

- TypeScript étant typé, il est nécessaire de définir les types de nos variables, paramètres et type de retour.

```
// Les declarations definissent un type
let str : string;
str = "Hello";
// En donnant une valeur ts definie automatiquement le type
let str2 = "Hello";

let i : number;
// Erreur une chaine de caracteres n'est pas un entier
i = "Hello";
```

TypeScript

```
// Lors de la definition des fonction, les parametres doivent etre typé
const add = (a:number, b:number) => {
  ... return a+b
}
// Un parametre avec une valeur par default n'a pas besoin de typage
// l'on peut specifier le type de retour
function sub (a:number, b=1): number{
  ... return a-b;
}
```

TypeScript

```
function sub(a:number, b=1): number{  
  ... return a-b;  
}
```

```
// l'on peut aussi typer en fonction  
// : ([paramName:paramType])=>typeRetour  
let foo: (a:number, b:number)=>number;  
foo = sub;
```

Typescript

- Il est possible de typer les tableaux pour assurer le type de tous leurs éléments.

```
// Definition des tableaux
let tab: number[];
tab.push(1)
// Impossible d'ajouter a un tableau d'entier un string
tab.push("Hello")

//Une autre manière de déclarer les tableaux
let array : Array<number>;
array.push(1)
array.push("Hello")
```

Typescript

Les classes :

- La déclaration des classes utilise l'identifiant 'class'
- Par convention indiscutable, Le nom des classes commence par une majuscule!!!
- On définit les attributs (équivalent des propriétés) en premier
- On peut définir un constructeur. Si ce n'est pas le cas, un constructeur sans paramètres sera créé implicitement.
- On peut définir des méthodes
- L'appel des attributs utilise 'this' comme en javascript.

```
class MaClass {  
    ... private att : number;  
  
    ... public constructor(){  
        ... this.att = 1;  
    }  
  
    ... public methode(param:any){  
        ... //todo  
    }  
}
```


TypeScript

Les classes :

- Il est possible de restreindre la visibilité des attributs, constructeurs et méthodes d'une classe
- Il existe 3 visibilités :
 - "public" : la propriété est accessible sans restrictions. C'est la visibilité par défaut.
 - "protected" : la propriété est accessible uniquement par les enfants
 - "private" : seuls les éléments de la classe peuvent y avoir accès

```
class MaClass {  
    ... private att : number;  
    ... public attPublic : any;  
    ... attSansVisibilite : any;  
    ... private attPrivate : any;  
  
    ... public constructor(){  
        ... this.att = 1;  
    }  
  
    ... public methode(param:any){  
        ... this.att = 1  
        ... this.attPublic = 1  
        ... this.attPrivate = 1  
        ... this.attSansVisibilite = 1  
    }  
}
```

```
const instance = new MaClass();  
console.log(instance.attPublic)  
console.log(instance.attSansVisibilite)  
console.log(instance.attPrivate) // N'est pas accessible car privé
```

TypeScript

Les classes :

- Les constructeurs sont des fonctions spéciales qui permettent l'instanciation d'une classe en objet.
 - On utilise le mot clé 'new' pour l'instanciation
 - Il ne peut y avoir qu'un seul constructeur pour une classe
 - Il se définit comme une fonction classique mais s'appelle 'constructor'
 - Il n'est pas forcément public
 - En général il permet l'initialisation des attributs de la classe
 - Il est possible d'utiliser un raccourci dans ses paramètres pour définir un attribut.

```
class MaClass2 {  
    constructor(public att="hello"){}}
```

```
const cls = new MaClass2();  
const cls2 = new MaClass2("bye");  
console.log(cls.att); //hello  
console.log(cls2.att); //bye
```

TypeScript

Les classes :

- Une fonction dans une classe est appelée méthode.
 - Elle a le fonctionnement classique d'une fonction

```
class MaClass2 {  
    ... constructor(private att:number){}  
    ... public methode(){  
    ...     console.log(this.att)  
    ... }  
    ...  
    ... public fmd = ()=> {  
    ...     console.log(this.att)  
    ... }  
}
```

Typescript

Les classes :

- Les éléments statiques sont communs à l'ensemble des instances de la classe et appelables directement par le nom de la classe où ils sont implémentés.

```
class MaClass {  
    ....// definition d'un attribut static  
    ....private static staticAtt = "Static";  
    ....public att = "att"  
  
    ....// definition d'une methode statis  
    ....public static smtd(){  
    ....    ....// Impossible d'appelé un attribut classique dans une methode statis  
    ....    console.log(this.att)  
    ....}  
    ....public mtd(){  
    ....    console.log(this.mtd)  
    ....}  
}
```

Typescript

L'héritage :

- Comme en javascript il est possible d'hériter d'une classe parent.
 - Il ne peut toujours pas y avoir d'héritage multiple (même s'il est possible de le "faire" avec du mixing object)

```
class Super {  
    ... public attSuper = ""  
    ... public methodeSuper(){  
    ... }  
}  
// Classe enfant heritant de la classe Super  
class Classe extends Super {  
    ... public attClasse = "";  
    ... public methodeClasse(){  
    ... }  
}
```

```
const inst = new Classe();  
inst.attClasse  
inst.attSuper  
inst.methodeClasse()  
inst.methodeSuper()
```

TypeScript

L'héritage :

- Les méthodes et attributs "private" ne sont pas hérités.

```
class Super {  
    ... private attSuper = ""  
    ... private methodeSuper() {  
    ... }  
}  
// Classe enfant heritant de la classe Super  
class Classe extends Super {  
    ... public attClasse = "";  
    ... public methodeClasse() {  
    ... // pas d'accès aux attributs private de la classe parent  
    ... this.attSuper  
    ... }  
}
```

```
const inst = new Classe();  
inst.attClasse  
// Attribut private  
inst.attSuper  
inst.methodeClasse()  
// Methode private  
inst.methodeSuper()
```

Typescript

L'héritage :

- Il est possible d'utiliser la visibilité "protected" qui rend visible aux enfants mais pas en dehors.

```
class Super {  
  ... protected attSuper = ""  
  ... protected methodeSuper(){  
    ...}  
}  
// Classe enfant heritant de la classe Super  
class Classe extends Super {  
  ... public attClasse = "";  
  ... public methodeClasse(){  
    ... // acces possibles aux methodes et attributs protected  
    ... this.attSuper  
    ... this.methodeSuper()  
  ...}  
}
```

```
const inst = new Classe();  
inst.attClasse  
// Attribut protected  
inst.attSuper  
inst.methodeClasse()  
// Methode protected  
inst.methodeSuper()
```

Typescript

Le polymorphisme :

- Typescript permettant le typage et l'héritage, il est possible d'utiliser le polymorphisme.

```
class Super {  
    ... public attSuper = ""  
    ... public methodeSuper(){  
    ... }  
}  
// Classe enfant heritant de la classe Super  
class Classe extends Super {  
    ... public attClasse = "";  
    ... public methodeClasse(){  
    ... // acces possibles aux methodes et attributs protected  
    ... this.attSuper  
    ... this.methodeSuper()  
    ... }  
}
```

```
const instClass:Classe = new Classe();  
instClass.attClasse  
instClass.attSuper  
instClass.methodeClasse()  
instClass.methodeSuper()
```

```
const instSuper:Super = new Classe();  
// Non definie dans super  
instSuper.attClasse  
instSuper.attSuper  
// Non definie dans Super  
instSuper.methodeClasse()  
instSuper.methodeSuper()
```


Typescript

Le polymorphisme :

- Il est possible de "override" des méthodes

```
class Super {  
  ... public attSuper = ""  
  ... public methode(){  
    ... console.log("SUPER")  
  }  
}  
// Classe enfant heritant de la classe Super  
class Classe extends Super {  
  ... public attClasse = "";  
  ... // Override de la methode 'methode' de Super  
  ... public methode(){  
    ... console.log("CLASSE")  
  }  
}
```

```
const sup = new Super()  
const classe = new Classe();  
const classeSuper: Super = new Classe()  
  
sup.methode() // affiche : SUPER  
classe.methode() // affiche : CLASSE  
classeSuper.methode() // affiche : CLASSE
```

TypeScript

Les interfaces :

- Il est possible de créer des interfaces
 - Elles définissent le prototype des méthodes
 - Toutes les méthodes doivent être publiques
 - On peut définir des mutateurs

```
interface Calculatrice {  
    ... add: (...x:number[])=>number;  
    ... mul: (...x:number[])=>number;  
    ... pow: (x:number, y:number)=>number;  
}
```

Typescript

Les interfaces :

- Elles sont implémentées dans une classe grâce au mot clé 'implements'
- Une classe implémentant une interface doit décrire toutes ses méthodes.

```
class MaCalculatrice implements Calculatrice {  
    ... public add = (...x: number[]) => {  
        ... return x.reduce((a,b)=>a+b,0)  
        ... }  
    ... public mul = (...x: number[]) => {  
        ... return x.reduce((a,b)=>a*b,0)  
        ... }  
    ... public pow = (x: number, y: number) => {  
        ... let result = x  
        ... for (let i=0; i<y; i++) result*=x  
        ... return result  
        ... };  
}
```

```
class MaCalculatrice2 implements Calculatrice {  
    ... public add = (...x: number[]) => {  
        ... let result = 0  
        ... for (let v of x) result+=v  
        ... return result  
        ... }  
    ... public mul = (...x: number[]) => {  
        ... let result = 0  
        ... for (let v of x) result*=v  
        ... return result  
        ... }  
    ... public pow = (x: number, y: number) => {  
        ... let result = x  
        ... for (let i=0; i<y; i++) result*=x  
        ... return result  
        ... };  
}
```

Typescript

Les interfaces :

- Elles peuvent servir de Type (Bonne pratique!!)

```
let calculatrice : Calculatrice;  
calculatrice = new MaCalculatrice()  
calculatrice.add(1,2,3)  
calculatrice.mul(1,2,3)  
calculatrice.pow(7,2)  
calculatrice = new MaCalculatrice2()  
calculatrice.add(1,2,3)  
calculatrice.mul(1,2,3)  
calculatrice.pow(7,2)
```

TypeScript

Les interfaces :

- Une classe peut implémenter plusieurs interfaces

```
interface I1 {  
    ... mth1: () => void  
}  
interface I2 {  
    ... mth2: () => void  
}  
class A implements I1, I2 {  
    ... mth1 = () => {};  
    ... mth2 = () => {};  
}
```

TypeScript

Les interfaces :

- Une interface peut hériter d'une ou plusieurs interfaces avec 'extends'
 - L'interface enfant aura alors les méthodes de l'interface parent.

```
interface I1 {  
  ... mth1: () => void  
}  
interface I2 extends I1 {  
  ... mth2: () => void  
}  
class A implements I2 {  
  ... mth1 = () => {};  
  ... mth2 = () => {};  
}
```

TypeScript

Les classes abstraites :

- Il est possible de créer des classes abstraites
 - Elles sont définies avec un modificateur 'abstract'
 - Elles ne peuvent pas être instanciées
 - Elles ressemblent aux classes classiques (dites concrètes)
 - Elles peuvent (facultatif) définir des méthodes abstraites avec 'abstract'

```
abstract class AbstractClass {  
    ... public abstract methode: () => void;  
    ... protected abstract foo: () => void;  
    ... private field: number;  
    ... constructor() {}  
}  
  
class ConcreteClass extends AbstractClass {  
    ... public methode = () => {}  
    ... public foo = () => {}  
}  
  
// Impossible car classe abstraite  
let ac1s = new AbstractClass();  
let cls: AbstractClass = new ConcreteClass();
```

TypeScript

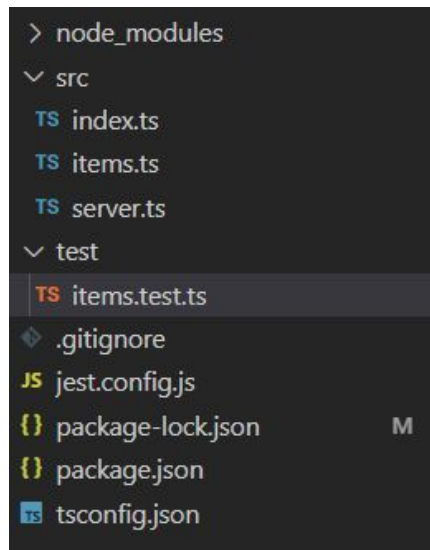
Les classes abstraites :

- Une classe Abstraite peut implémenter une interface.
 - Elle peut soit définir la méthode soit la rendre abstraite.

```
interface MonInterface {  
    ... foo : () => void  
}
```

```
abstract class MaClassAbstraite implements MonInterface {  
    ... abstract foo : () => void;  
}
```


TypeScript



tsconfig.json :

```
{
  "compilerOptions": {
    "module": "commonjs",
    "esModuleInterop": true,
    "outDir": "dist",
    "target": "es6",
    "strict": true
  },
  "include": [
    "src/**/*"
  ]
}
```

Typescript

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

export default class Server {
  readonly port: number

  constructor (port: number) {
    this.port = port
  }

  start () {
    const app = express()
    app.get('/', function (req: Request, res: Response) {
      res.send('Salut les gens')
    })
    app.listen(this.port, function () {
      console.log('Serveur démarré')
    })
  }
}
```

Exercice

Et on réécrivait notre application de film en Typescript ?

Tests unitaires

- Framework : Jest
- `npm install --save-dev jest` (Typescript ajouter `@types/jest`)

sum.js :

```
function sum(a, b) {  
  return a + b;  
}  
module.exports = sum;
```

sum.test.js :

```
test('adds 1 + 2 to equal 3', () => {  
  expect(sum(1, 2)).toBe(3);  
});
```

Tests unitaires

```
export default class Items {  
  static getFirstItem () {  
    return 'Item1'  
  }  
}
```

```
import Items from '../src/items';  
  
describe('Items', function () {  
  it('should return the first item', function () {  
    expect(Items.getFirstItem()).toBe('Item1');  
  })  
})
```

Exercice

Tester 2 fonctions/méthodes de votre code

Passport.js

- Passport :un middleware Node.js qui offre une variété de stratégies d'authentification de requête différentes faciles à mettre en œuvre. Par défaut, il stocke l'objet utilisateur en session.

Passport.js

installation :

```
npm install --save passport passport-local mongoose
```

pour typescript ajouter préfixe : @types/

Passport.js

db.js

```
import mongoose from 'mongoose'

const dbConnection = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGODBURI)
    console.log(`Db runs in ${conn.connection.host}`)
  } catch {
    process.exit(1)
  }
}

export default dbConnection
```

Passport.js

models/User.js

```
import mongoose from 'mongoose'

const userSchema = mongoose.Schema({
  username: { type: 'string', required: true },
  email: { type: 'string', required: true, unique: true },
  password: { type: 'string', required: true },
  registrationDate: { type: Date, default: Date.now }
})

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

Passport.js

views/register.ejs

```
<div class="register">
  <form autocomplete="off" action="/users/register" method="post">

    <label for="username">Username: </label><br>
    <input type="text" name="username" id="username" required ><br><br>

    <label for="email">Email: </label><br>
    <input type="email" name="email" id="email" required ><br><br>

    <label for="pwd">Password: </label><br>
    <input type="password" name="pwd" id="pwd" required ><br><br>

    <label for="pwdConf">Confirm Password: </label><br>
    <input type="password" name="pwdConf" id="pwdConf" required ><br><br>

    <button>Register</button>

  </form>
</div>
```

Passport.js

routes/users.js

```
router.post('/register', async (req, res) => {
  const { username, email, pwd, pwdConf } = req.body

  // VALIDATION
  const errors = []

  if(pwd !== pwdConf) errors.push(`Passwords don't match`)

  const emailTaken = await User.findOne({ email })
  if(emailTaken) errors.push(`Email taken!`)

  if(errors.length > 0) res.redirect('/register', { errorMessage: errors})

  // REGISTRATION
  const hashpwd = await bcrypt.hash(pwd, 12)
  let user = new User({ username, email, password: hashpwd })

  try {
    await user.save()
    res.redirect('/users/login')
  } catch {
    res.redirect('/', { message: "There was a problem register the user" })
  }
})
```

Passport.js

routes/users.js

```
router.post('/login', passport.authenticate('local', {  
  successRedirect: '/',  
  failureRedirect: '/users/register'  
}))
```

Passport.js

passport.js

```
import local_strategy from "passport-local"
import bcrypt from "bcryptjs"
import User from './models/User.js'

const LocalStrategy = local_strategy.Strategy

const customFields = {
  usernameField: 'email',
  passwordField: 'password',
}

const verifyCallback = (email, password, done) => {

  User.findOne({ email })
    .then (user => {
      if(!user){ return done(null, false) }
      else {
        const isValid = bcrypt.compareSync(password, user.password)
        if(!isValid){
          return done(null, false)
        }else{
          return done(null, user)
        }
      }
    })
}
```

Passport.js

passport.js

```
const authenticateUser = (passport) => {  
  
  passport.use(new LocalStrategy(customFields, verifyCallback))  
  
  passport.serializeUser((user, done) => done(null, user.id))  
  
  passport.deserializeUser( (id, done) => User.findById(id)  
    .then( user => done(null, user))  
    .catch( err => done(err))  
  )  
  
}  
  
export default authenticateUser
```

Passport.js

app.js

```
import dotenv from 'dotenv'
import passport from 'passport'
import express from 'express'
import ejsLayout from 'express-ejs-layouts'
import methodOverride from 'method-override'
import session from 'express-session'
import MongoStore from 'connect-mongo'

import authenticateUser from './passport.js'
import userRoutes from './routes/users.js'
import indexRoute from './routes/index.js'
import db from './db.js'

const app = express()
|
// use environment variables locally.
if(process.env.NODE_ENV !== 'production') dotenv.config()
db()
authenticateUser(passport)

// register templating engine
app.set('view engine', 'ejs')
app.use(ejsLayout)

// receive form data
app.use(express.urlencoded({extended: false, limit: '2mb'}))
```


Passport.js

app.js

```
// use express sessions
app.use(session({
  secret: process.env.SECRET,
  resave: false,
  saveUninitialized: false,
  store: MongoStore.create({ mongoUrl: process.env.MONGODBURI }),
  cookie: {
    maxAge: 1000 * 60 * 60 * 24
  }
}))

// passport session
app.use(passport.initialize())
app.use(passport.session())
app.use(methodOverride('_method'))

// routes
app.use('/', indexRoute)
app.use('/users', userRoutes)

// listen for requests
const PORT = process.env.PORT || 5000
app.listen(PORT, console.log(`Listening on port ${PORT}`))
```

Passport.js

routes/index.js

```
import express from "express"

const router = express.Router()

router.get('/', (req, res) => res.render('index'))

export default router
```

Passport.js

- JSON Web Tokens est une norme d'authentification qui fonctionne en attribuant et en transmettant un jeton crypté dans les demandes qui aide à identifier l'utilisateur connecté, au lieu de stocker l'utilisateur dans une session sur le serveur et de créer un cookie.

Passport.js

passport.js :

Ajout de la JWTStrategy

```
import local_strategy from "passport-local"
import bcrypt from "bcryptjs"
import User from './models/User.js'
import passportJWT from 'passport-jwt';

const JWTStrategy = passportJWT.Strategy;
const ExtractJWT = passportJWT.ExtractJwt;
const LocalStrategy = local_strategy.Strategy
```

```
const authenticateUser = (passport) => {

  passport.use(new LocalStrategy(customFields, verifyCallback))

  passport.serializeUser((user, done) => done(null, user.id))

  passport.deserializeUser( (id, done) => User.findById(id)
    .then( user => done(null, user))
    .catch( err => done(err))
  )

  passport.use(new JWTStrategy({
    jwtFromRequest: ExtractJWT.fromAuthHeaderAsBearerToken(),
    secretOrKey : 'your_jwt_secret'
  }),
  function (jwtPayload, cb) {

    //find the user in db if needed. This functionality may be omitted if you store everything you'll need in JWT payload.
    return UserModel.findOneById(jwtPayload.id)
      .then(user => {
        return cb(null, user);
      })
      .catch(err => {
        return cb(err);
      });
  });
};
```

Passport.js

routes/user.js :

Créer un fichier de route pour les routes protégées.

Implémenter l'action de connexion et gérer les erreurs.

```
import jwt from 'jsonwebtoken';
const router = express.Router();

// LOGIN
router.get('/login', (req, res) => res.render('login'));

router.post('/login', function (req, res, next) {
  passport.authenticate('local', {session: false}, (err, user, info) => {
    if (err || !user) {
      return res.status(400).json({
        message: 'Something is not right',
        user : user
      });
    }
    req.login(user, {session: false}, (err) => {
      if (err) {
        res.send(err);
      }
      // generate a signed son web token with the contents of user object and return it in the response
      const token = jwt.sign(user.toJSON(), 'your_jwt_secret');
      user = {
        username: user.username,
        email: user.email
      }
      return res.json({user, token});
    });
  })(req, res);
});
```

Passport.js

routes/register.js :

Créer une route distincte pour pouvoir s'inscrire.

```
router.post('/register', async (req, res) => {
  const { username, email, pwd, pwdConf } = req.body
  // VALIDATION
  let errors = [];
  console.log(req.body);
  if (typeof pwd === 'undefined') errors.push('Password required');
  if (pwd !== pwdConf) errors.push('Passwords don't match');

  const emailTaken = await User.findOne({ email })
  if(emailTaken) errors.push('Email taken!')

  if(errors.length > 0) {
    res.send(errors)
  } else {
    // REGISTRATION
    const hashpwd = await bcrypt.hash(pwd, 12)
    let user = new User({ username, email, password: hashpwd })
    try {
      await user.save()
      res.send('Register success')
    } catch {
      res.send('Wrong')
    }
  }
});
```

Passport.js

app.js :

Ajout du format json:

```
app.use(express.json())
```

Protéger une route :

```
app.use('/', passport.authenticate('jwt', {session: false}), indexRoute);
```