Developpement Web

Chapitre 5: JavaScript/TypeScript en Back-End

M. Mohammed BELATAR

m.belatar@emsi.ma

avec Mme. FZ MOUTAI et M. YF EBOBISSE

3ème Année Ingénierie en Informatique et Réseaux

28 décembre 2024

Sommaire

Qu'est-ce que Node.js ?

- Node.js est un environnement d'exécution pour JavaScript basé sur le moteur V8 de Chrome.
- Permet d'exécuter du JavaScript côté serveur.
- Principaux avantages :
  - Performance élevée grâce à une architecture non bloquante et orientée événements.
  - Utilisation d'un seul langage, JavaScript/TypeScript, pour le front-end et le back-end.
- Cas d'utilisation :
  - APIs REST, applications en temps réel, systèmes de streaming, etc.

Exemple : Création d'un simple serveur HTTP.

Exemple : Serveur HTTP avec Node.js

Code Node.js (server.js)

```
 const http = require('http');
const server = http.createServer((req, res) => {
   res.statusCode = 200;
   res.setHeader('Content-Type', 'text/plain');
   res.end('Bonjour, Node.js !');
});
server.listen(3000, () => {
   console.log('Serveur démarré sur http://localhost:3000');
});
```

Exécution :

- Lancez le serveur : `node server.js`.
- Accédez à `http://localhost:3000` dans votre navigateur.

Installation de Node.js et npm

- **Node.js** inclut npm (Node Package Manager).
- Télécharger et installer Node.js depuis nodejs.org.
- Vérifiez l'installation :

Commandes

```
node --version    // Affiche la version de Node.js.
npm --version     // Affiche la version de npm.
```

Pourquoi utiliser npm ?

- Gère les dépendances des projets.
- Permet d'installer, mettre à jour, et supprimer des packages JavaScript/TypeScript.

Qu'est-ce qu'un Module ?

- Un module est une unité de code réutilisable et indépendante.
- Permet d'organiser le code en fichiers séparés.
- Node.js supporte deux types de modules :
  - **CommonJS** : Système de modules natif de Node.js (dépassé).
  - **ECMAScript Modules (ESM)** : Standard moderne.

Avantages des Modules :

- Simplifie la maintenance et la collaboration.
- Facilite la réutilisation du code dans différents projets.

Modules dans Node.js (CommonJS)

- Node.js utilise CommonJS par défaut pour les modules.
- Un module exporte ses fonctionnalités avec `module.exports`.
- Les fonctionnalités exportées sont importées avec `require`.

Exemple : Module Personnalisé

```
// fichier: utils.js
function addition(a, b) {
  return a + b;
}
module.exports = { addition };
```

Utilisation dans un autre fichier :

Exemple : Importer le Module

```
const utils = require('./utils');
console.log(utils.addition(2, 3)); // Affiche 5
```

Modules ECMAScript (ESM)

- ESM est le système de modules standardisé par JavaScript (depuis ES6).
- Utilise `export` pour exporter des fonctionnalités.
- Utilise `import` pour les importer.

Exemple : Module ESM

```
// fichier: utils.mjs
export function addition(a, b) {
  return a + b;
}
```

Utilisation dans un autre fichier :

Exemple : Importer le Module

```
import { addition } from './utils.mjs';
console.log(addition(2, 3)); // Affiche 5
```

Initialisation d'un Projet avec npm

- Créez un nouveau projet avec la commande :

Commande

```
npm init
```

- Suivez les instructions pour configurer le fichier `package.json`.
- Exemple de fichier `package.json` minimal :

Exemple : package.json

```
{   "name": "mon-projet",
 "version": "1.0.0",
 "description": "Un exemple de projet Node.js",
 "main": "index.js",
 "scripts": {     "start": "node index.js"   },
 "dependencies": { },
 "devDependencies": { } }
```

Installer des Packages avec npm

- Installer une dépendance (exemple : `express`).

Commande

```
npm install express
```

- Ajouter des dépendances de développement (exemple : `nodemon`).

Commande

```
npm install --save-dev nodemon
```

- Supprimer un package.

Commande

```
npm uninstall express
```

Note : npm met automatiquement à jour le fichier `package.json`.

Exemple : Utiliser un Package npm avec ESM

Création d'un Serveur avec Express

```
 import express from 'express';
const app = express();
app.get('/', (req, res) => {
   res.send('Bienvenue avec Express et TypeScript !');
});
app.listen(3000, () => {
   console.log('Serveur démarré sur http://localhost:3000');
});
```

Étapes :

- Installez `express` : `npm install express`.
- Exécutez le fichier avec : `node server.mjs`.

Mise à Jour des Packages avec npm

- Vérifiez les mises à jour disponibles.

Commande

```
npm outdated
```

- Mettez à jour un package spécifique.

Commande

```
npm update nom-du-package
```

- Mettez à jour toutes les dépendances.

Commande

```
npm update
```

Résumé : Node.js et npm

- Node.js permet d'exécuter JavaScript côté serveur.
- npm facilite la gestion des dépendances dans les projets.
- Nous avons vu :
  - Les notions de modules et leur utilisation (ESM).
  - Comment installer et configurer Node.js.
  - Comment utiliser npm pour initialiser un projet et gérer les packages.

Sommaire

Pourquoi utiliser TypeScript avec Node.js ?

- TypeScript apporte un typage statique, réduisant les erreurs de développement.
- Facilite la maintenance et la lisibilité du code dans des projets complexes.
- Intègre des outils modernes comme **IntelliSense**, l'auto-complétion et les vérifications statiques.
- Idéal pour développer des API robustes et évolutives.

Étape 1 : Installer TypeScript et les outils nécessaires

- Installez TypeScript, `ts-node` et les types de Node.js :

Commandes

```
npm install typescript ts-node @types/node --save-dev
```

- Générez un fichier `tsconfig.json` pour configurer TypeScript :

Commande

```
npx tsc --init
```

Résultat : Un fichier `tsconfig.json` est créé pour gérer les paramètres du compilateur TypeScript.

Étape 2 : Configurer tsconfig.json

- Configuration typique pour un projet Node.js avec TypeScript :

Exemple : tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES6", // Version de JavaScript générée
    "module": "ESNext", // Utilise les modules ESM
    "rootDir": "./src", // Dossier contenant les fichiers .ts
    "outDir": "./dist", // Dossier de sortie des fichiers
compilés
    "strict": true, // Active les vérifications strictes
    "esModuleInterop": true, // Compatibilité avec CommonJS
    "skipLibCheck": true // Ignore la vérification des
bibliothèques
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

Étape 3 : Structure du projet

- Organisez votre projet en deux dossiers :
  - `src/` : Contient vos fichiers TypeScript (`index.ts`).
  - `dist/` : Contient les fichiers JavaScript compilés.
- Exemple d'arborescence :

Arborescence du Projet

```
projet-node-typescript/
   src/
     index.ts
   dist/
     index.js
   package.json
   tsconfig.json
```

Étape 4 : Exemple de fichier TypeScript (src/index.ts)

- Un exemple simple d'utilisation de TypeScript avec Express.js :

Exemple : index.ts

```
 import express from 'express';
const app = express();
const port = 3000;
// Middleware pour parser JSON
app.use(express.json());
// Route GET
app.get('/', (req, res) => {
   res.send('Bienvenue dans TypeScript avec Node.js et
Express!');
});
// Démarrer le serveur
app.listen(port, () => {
   console.log(`Serveur démarré sur http://localhost:${port}`);
});
```

Étape 5 : Configurer les scripts npm

- Ajoutez les scripts suivants dans le fichier `package.json` :

Exemple : `package.json`

```
 "scripts": {
  "build": "tsc", // Compile TypeScript en JavaScript
  "start": "node dist/index.js", // Exécute le fichier compilé
  "dev": "node --loader ts-node/esm src/index.ts", // Exécute
TypeScript directement en mode dev
}
```

Avantage : Simplifie les commandes pour développer et exécuter le projet.

Configuration alternative mode dev

```
 "scripts": {
  "dev": "nodemon --watch src --exec node --loader ts-node/esm
src/index.ts" }
```

Étape 6 : Compilation et Exécution

- **Mode production :**
  1. Compilez les fichiers TypeScript : `npm run build`.
  2. Exécutez le fichier compilé : `npm start`.
- **Mode développement :**
  1. Exécutez directement TypeScript avec `ts-node` : `npm run dev`.

Résumé

- Nous avons configuré un projet Node.js avec TypeScript.
- Bénéfices principaux :
  - Typage statique pour un code robuste.
  - Auto-complétion et détection des erreurs à la compilation.
  - Simplification du développement avec `ts-node`.
- Prochaines étapes :
  - Développer des APIs REST avec Express et TypeScript.
  - Gérer des bases de données avec TypeORM.

Sommaire

Qu'est-ce qu'Express.js ?

- Express.js est un framework minimaliste pour créer des serveurs web en Node.js.
- Utilisé pour construire des APIs REST, des middlewares, et gérer les routes.
- Express est flexible, léger, et largement adopté.

Avantages avec TypeScript :

- Gestion des types pour les requêtes, réponses et middlewares.
- Réduction des erreurs grâce au typage statique.

Installation et Configuration d'Express.js

- Installez les dépendances nécessaires :

Commandes

```
npm install express
npm install --save-dev @types/express
```

- Créez un fichier TypeScript principal (`index.ts`).
- Exemple de configuration d'un serveur Express.js :

Exemple : Serveur Express.js

```
 import express, { Request, Response } from 'express';
const app = express();
app.use(express.json());
app.get('/', (req: Request, res: Response) => {
   res.send('Bonjour depuis Express avec TypeScript!');
});
app.listen(3000, () => {
   console.log('Serveur démarré sur http://localhost:3000');
});
```

Exemple d'API REST (GET, POST)

- Ajoutez des routes pour gérer des requêtes GET et POST.
- Exemple d'une API REST simple :

Exemple : API REST

```
 app.get('/users', (req: Request, res: Response) => {
   res.json([{ id: 1, name: 'Aïcha'}, { id: 2, name: 'Hamid'}]);
});
app.post('/users', (req: Request, res: Response) => {
   const user = req.body;
   res.status(201).json(user);
});
```

Testez l'API : Utilisez des outils comme `Postman` ou `curl`.

Servir des ressources statiques en Node.js avec Express

- Les ressources statiques incluent les fichiers HTML, CSS, JavaScript côté client, images, etc.
- Express permet de servir facilement des fichiers statiques.

Configuration

```
 import express from "express";
const app = express();
// Définir le dossier public contenant les fichiers
statiques
app.use(express.static("./public"));
app.listen(3000, () => {
   console.log("Serveur démarré sur http://localhost:3000");
});
```

Avantages :

- Simple à configurer.
- Compatible avec les fichiers HTML existants.

Exemple de structure pour les fichiers statiques

Structure de projet :

- `/src/index.ts` : Fichier principal du serveur.
- `/public/` : Contient les fichiers statiques.
- Exemple :

Structure

```
/public
 |- index.html
 |- styles.css
 |- app.js
```

Résultat :

- Accéder à `http://localhost:3000/index.html`.
- Les fichiers CSS et JS sont automatiquement servis.

Introduction aux moteurs de templates

- Les moteurs de templates permettent d'intégrer des données dynamiques dans les pages HTML.
- Fonctionnent comme `<?php echo ... ?>` en PHP.

Exemples de moteurs de templates :

- **EJS** : Syntaxe proche du HTML classique.
- **Pug** : Syntaxe concise et puissante.
- **Handlebars** : Simplicité et extensibilité.

Configurer et utiliser EJS

- Installer le moteur de templates EJS :

Commande

```
npm install ejs
```

- Configurer Express pour utiliser EJS :

Configuration

```
 import express from "express";
const app = express();
// Définir EJS comme moteur de rendu
app.set("view engine", "ejs");
app.set("views", "./views");
app.get("/", (req, res) => {
   res.render("index", { title: "Bienvenue", message:
"Bonjour depuis EJS!" });
});
app.listen(3000,()=>{console.log("Serveur démarré");});
```

Créer un fichier de modèle avec EJS

Structure du projet :

- `/views/` : Contient les fichiers EJS.
- Exemple de fichier `index.ejs` :

Exemple : index.ejs

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

Résultat : La page affiche "Bonjour depuis EJS !" avec le titre "Bienvenue".

Exemple Avancé avec EJS

Ajout d'une liste dynamique :

Exemple : index.ejs

```
 <!DOCTYPE html> <html lang="fr"> <head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title><%= title %></title> </head>
<body>
  <h1><%= message %></h1>
  <ul>
    <% items.forEach(item => { %>
      <li><%= item %></li>
    <% }); %>
  </ul>
</body>
</html>
```

Passer des données dynamiques au modèle

Exemple de route avec données dynamiques :

Exemple : Express

```
app.get("/", (req, res) => {
  const data = {
    title: "Liste des éléments",
    message: "Bienvenue sur notre page!",
    items: ["Article 1", "Article 2", "Article 3"]
 };
  res.render("index", data);
});
app.listen(3000, () => { console.log("Serveur démarré"); });
```

Explication :

- La clé `items` est utilisée pour générer la liste dynamique.
- Les données passées à `res.render()` sont accessibles dans le fichier EJS.

Syntaxes spécifiques dans EJS

- <%= ... %> : Insère une valeur échappée (sécurisée contre XSS).
- <%- ... %> : Insère du contenu brut (non échappé).
- <% ... %> : Exécute du JavaScript sans insérer de contenu.

Exemples

```
 <h1><%= message %></h1>
<ul>
  <% items.forEach(item => { %>
    <li><%= item %></li>
  <% }); %>
</ul>  <!-- Exécute une boucle pour générer une liste -->
<div><%- "<b>HTML brut</b>" %></div>  <!-- Insère un contenu
HTML brut -->
```

Attention : Utilisez <%- ... %> avec précaution pour éviter les failles XSS.

Résultat attendu

Données

```
 title: "Liste des éléments",
message: "Bienvenue sur notre page!",
items: ["Article 1", "Article 2", "Article 3"]
```

Rendu HTML

```
 <!DOCTYPE html> <html lang="fr"> <head>
  <meta charset="UTF-8">
  <title>Liste des éléments</title> </head>
<body>   <h1>Bienvenue sur notre page!</h1>   <ul>
    <li>Article 1</li>
    <li>Article 2</li>
    <li>Article 3</li>
  </ul>
</body> </html>
```

Conclusion : EJS permet de facilement générer du contenu dynamique à partir des données+templates.

Alternatives à EJS

- **Pug** : Syntaxe concise et puissante.
  - Exemple : `p Bonjour depuis Pug!`
- **Handlebars** : Simplicité et extensibilité.
  - Exemple : `{{message}}`

Recommandation :

- Utilisez EJS pour une syntaxe proche du HTML.
- Privilégiez Pug ou Handlebars pour des projets nécessitant plus de flexibilité.

Sommaire

Introduction à mysql2

- **mysql2** est une bibliothèque Node.js pour interagir avec les bases de données MySQL.
- Permet d'exécuter des requêtes SQL brutes pour un contrôle total.
- Recommandée pour des projets simples ou nécessitant une optimisation fine.

Installation :

Commande

```
npm install mysql2
```

Connexion et Requêtes avec mysql2

- Étape 1 : Créez une connexion à la base de données.
- Étape 2 : Exécutez des requêtes SQL pour manipuler les données.

Exemple : Connexion et Insertion

```
 import mysql from "mysql2";
const dbConnection = mysql.createConnection({
   host: "localhost",
   user: "root",
   password: "password",
   database: "testDb" });
dbConnection.query(
   "INSERT INTO users (name, email) VALUES (?, ?)",
   ["Hassan", "hassan@example.com"],
   (err, results) => {
     if (err) throw err;
     console.log("Utilisateur ajouté :", results); } );
```

Avantage : Contrôle total sur les requêtes SQL.

API GET avec mysql2 et Express

Exemple : API GET

```
 import express from "express";
import mysql from "mysql2";
const app = express();
const dbConnection = mysql.createConnection({
   host: "localhost",
   user: "root",
   password: "password",
   database: "testDb" });
app.get("/users", (req, res) => {
   dbConnection.query("SELECT * FROM users", (err, rows) => {
     if (err) return res.status(500).send(err);
     res.json(rows);
   }); });
app.listen(3000, () => { console.log("Serveur ON"); });
```

Résultat : Retourne les utilisateurs sous forme JSON.

Introduction à TypeORM

- **TypeORM** est un ORM (Object Relational Mapping) pour Node.js compatible avec TypeScript.
- Permet de manipuler des bases relationnelles via des entités et des méthodes.
- Supporte plusieurs bases de données : MySQL, PostgreSQL, SQLite, etc.

Installation :

Commande

```
npm install typeorm reflect-metadata mysql2
```

Configuration : Créez un fichier `ormconfig.json`.

Configuration TypeORM

- `ormconfig.json` est un fichier de configuration utilisé par TypeORM pour centraliser les paramètres de la base de données.
- Il est automatiquement détecté lorsque vous utilisez `createConnection()`.

Fichier de configuration : ormconfig.json

```
{
 "type": "mysql",
 "host": "localhost",
 "port": 3306,
 "username": "root",
 "password": "password",
 "database": "testDb",
 "entities": ["src/entity/*.ts"],
 "synchronize": true
}
```

Utilisation de ormconfig.json avec TypeORM

Configurer et établir une connexion avec la base de données

- Exemple de code pour établir une connexion avec TypeORM :

Exemple : Connexion avec createConnection()

```
 import "reflect-metadata";
import { createConnection } from "typeorm";
createConnection().then(async connection => {
   console.log("Connexion établie avec la base de données
!");
}).catch(error => console.log("Erreur de connexion :",
error));
```

Note : Placez le fichier ormconfig.json à la racine du projet pour qu'il soit détecté automatiquement.

Modèle TypeORM

- Définissez des entités représentant les tables dans src/entity/

Exemple d'entité "user.ts" : Modèle Utilisateur

```
 import { Entity, PrimaryGeneratedColumn, Column } from
"typeorm";
@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;
  @Column()
  name: string;
  @Column()
  email: string;
}
```

API REST avec TypeORM et Express

Utilisation de `ormconfig.json` pour la configuration

Exemple : API REST avec TypeORM

```
 import express from "express";
import  DataSource  from "typeorm";
import  User  from "./entity/User";
const app = express();
app.use(express.json());
const appDataSource = new DataSource(); // Utilise ormconfig.json
par défaut
appDataSource.initialize().then(() => {
  const userRepo = appDataSource.getRepository(User);
  app.post("/users", async (req, res) => {
    const user = userRepo.create(req.body);
    await userRepo.save(user);
    res.status(201).json(user);
  });
  app.listen(3000, () => { console.log("Serveur ON"); }); });
```

Recherche et Listing des Utilisateurs

Utilisation de TypeORM et Express

Exemple : Recherche et Listing

```
 appDataSource.initialize().then(() => {
  const userRepo = appDataSource.getRepository(User);
  // Récupérer tous les utilisateurs
  app.get("/users", async (req, res) => {
    const users = await userRepo.find();
    res.json(users);
  });
  // Rechercher un utilisateur par ID
  app.get("/users/:id", async (req, res) => {
    const user = await userRepo.findOneBy({ id:
parseInt(req.params.id) });
    if (!user) {
      return res.status(404).send("Utilisateur non trouvé");
    }
    res.json(user);
  });
});
```

Recherche et Listing : Détails et Explications

Utilisation des méthodes TypeORM pour les opérations de lecture

- **Récupération de tous les utilisateurs** :
  - Utilisation de `find()` : Récupère tous les enregistrements de la table `User`.
  - Résultat formaté automatiquement en JSON pour le client.
- **Recherche d'un utilisateur par ID** :
  - Utilisation de `findOneBy()` : Recherche un enregistrement unique correspondant à une condition.
  - Gestion des erreurs : Si aucun utilisateur n'est trouvé, retourne un statut HTTP 404.

Note : Ces méthodes garantissent une gestion propre des données.

Modification et Suppression des Utilisateurs

Utilisation de TypeORM et Express

Exemple : Modification et Suppression

```
 appDataSource.initialize().then(() => {
  const userRepo = appDataSource.getRepository(User);
  app.put("/users/:id", async (req, res) => {
    const user = await userRepo.findOneBy({ id:
parseInt(req.params.id) });
    if (!user) { return res.status(404).send("Utilisateur non
trouvé"); }
    userRepo.merge(user, req.body);
    await userRepo.save(user);
    res.json(user); });
  app.delete("/users/:id", async (req, res) => {
    const result = await userRepo.delete({ id:
parseInt(req.params.id) });
    if (result.affected === 0) {
      return res.status(404).send("Utilisateur non trouvé"); }
    res.status(204).send(); }); });
```

Modification et Suppression : Détails et Explications

Utilisation des méthodes TypeORM pour les opérations d'écriture et suppression

- **Modification d'un utilisateur** :
  - `merge()` : Combine l'objet existant avec les nouvelles données du corps de la requête.
  - `save()` : Met à jour ou insère l'objet modifié dans la base de données.
- **Suppression d'un utilisateur** :
  - `delete()` : Supprime l'utilisateur correspondant à l'ID fourni.
  - Vérification du résultat avec `result.affected` pour confirmer la suppression.

Résultat : Gère les modifications et suppressions en respectant les bonnes pratiques.

API REST avec TypeORM et Express

Solution alternative sans `ormconfig.json`

Exemple : Déclaration inline des infos de BDD

```
import express from "express";
import { DataSource } from "typeorm";
import { User } from "./entity/User";
const app = express();
app.use(express.json());
const appDataSource = new DataSource({
  type: "mysql",
  host: "localhost",
  username: "root",
  password: "password",
  database: "testDb",
  entities: [User],
  synchronize: true });
appDataSource.initialize().then(() => { ... });
  app.listen(3000, () => { console.log("Serveur ON"); });
});
```

Autres ORM pour Node.js

- **Prisma** :
  - ORM moderne avec migrations automatiques.
  - Génère un client TypeScript pour interagir avec la base.
- **Sequelize** :
  - API flexible pour SQL brut et ORM.
  - Moins intégré à TypeScript que TypeORM.
- **Mongoose** :
  - Conçu pour MongoDB.
  - Manipulation facile des documents JSON.

Sommaire

Qu'est-ce que Lodash ?

- **Lodash** est une bibliothèque JavaScript utilitaire qui simplifie les manipulations complexes de données.
- Fournit des fonctions pour :
  - Manipuler les tableaux, objets, chaînes de caractères.
  - Faciliter la programmation fonctionnelle (ex. : `debounce`, `throttle`).
- Compatible avec Node.js et les projets TypeScript.

Pourquoi utiliser Lodash avec TypeScript ?

- Intégration complète grâce aux définitions de types `@types/lodash`.
- Sécurité et autocomplétion des types dans les éditeurs.

Installation de Lodash et Types

- Installez Lodash :

Commande

```
npm install lodash
```

- Installez les types TypeScript pour Lodash :

Commande

```
npm install --save-dev @types/lodash
```

- Exemple d'import :

Code

```
import _ from "lodash";
```

Manipuler des Tableaux avec Lodash

- Lodash fournit des méthodes puissantes pour filtrer, transformer ou réduire des tableaux.

Exemple : Trouver les nombres pairs

```
 import _ from "lodash";
const numbers: number[] = [1, 2, 3, 4, 5, 6];
const evenNumbers: number[] = _.filter(numbers, n => n % 2 ===
0);
console.log(evenNumbers); // [2, 4, 6]
```

Résultat : Affiche les nombres pairs dans le tableau.

Clonage et Manipulation d'Objets

- Clonage profond d'un objet avec Lodash.
- Assure que les sous-objets sont également clonés (pas de référence partagée).

Exemple : Clonage profond

```
 interface User {
  name: string;
  details: { age: number; city: string; };
}
const user: User = { name: "Aïcha", details: { age: 25, city:
"Rabat" } };
const clonedUser: User = _.cloneDeep(user);
console.log(clonedUser);
```

Résultat : L'objet est entièrement cloné sans partager de références.

Manipulation des Chaînes de Caractères

- Lodash offre des fonctions pour transformer les chaînes.
- Exemple : Convertir une chaîne en camelCase.

Exemple : camelCase

```
 const text: string = "Bonjour le monde";
const camelCaseText: string = _.camelCase(text);
console.log(camelCaseText); // bonjourLeMonde
```

Résultat : Convertit la chaîne en `camelCase`.

Programmation Fonctionnelle avec Lodash

- Fonctions comme `debounce` pour limiter l'exécution d'une fonction.
- Pratique pour les événements comme `scroll`, `click`.

Exemple : Utilisation de debounce

```
 const handleClick: () => void = _.debounce(() => {
  console.log("Clicked!");
}, 300);
window.addEventListener("click", handleClick);
```

Résultat : Limite les appels fréquents de la fonction.

Avantages de Lodash avec TypeScript

- Intégration avec TypeScript pour :
  - Vérification des types et autocomplétion.
  - Documentation intégrée via les définitions de types.
- Lodash simplifie la manipulation des données complexes.

Alternatives natives : Préférez les fonctions JavaScript modernes (`filter`, `map`) si elles suffisent.

Ressources et Références

- **Documentation officielle :**
  - Node.js : https://nodejs.org/
  - TypeScript : https://www.typescriptlang.org/
  - Express.js : https://expressjs.com/
  - TypeORM : https://typeorm.io/
- **Guides et tutoriels :**
  - MDN Web Docs (JavaScript) : https://developer.mozilla.org/fr/docs/Web/JavaScript
  - FreeCodeCamp : https://www.freecodecamp.org/
- **Livres recommandés :**
  - "Node.js Design Patterns" par Mario Casciaro et Luciano Mammino.
  - "TypeScript Quickly" par Yakov Fain et Anton Moiseev.
- **Outils pratiques :**
  - Postman : https://www.postman.com/
  - Lodash Documentation : https://lodash.com/