Guide Complet pour Débutants : Développer une Application Full Stack avec Node.js et React

To-Do App

Réalisé par :

Nourhene Abbes et Yosr Joulek

Table des matières

Introduction Générale

Dans un contexte où les applications web dynamiques et interactives sont devenues la norme, la maîtrise du développement backend est essentielle pour tout développeur souhaitant construire des solutions complètes, robustes et évolutives.

Ce rapport a pour objectif d'explorer en profondeur les bases du développement backend avec Node.js et Express.js, tout en intégrant une base de données MongoDB via Mongoose, et en assurant une sécurité minimale avec JWT et bcrypt. Pour illustrer l'ensemble des notions abordées, un projet concret de To-Do List a été développé sous forme d'une API RESTful complète.

Afin de couvrir tout le cycle de développement web moderne, une partie frontend a également été réalisée à l'aide de React.js, démontrant l'interconnexion entre le backend et l'interface utilisateur via des requêtes HTTP sécurisées.

Ce document s'adresse à la fois aux étudiants et aux développeurs souhaitant acquérir une vision claire, structurée et progressive du développement web avec les technologies JavaScript les plus populaires.

Chapitre I

Introduction

Node.js est un environnement d'exécution JavaScript basé sur le moteur V8 de Google Chrome. Il permet d'exécuter du JavaScript côté serveur (et non seulement dans le navigateur).

I Avantages :

- Rapide et asynchrone grâce à l'event loop
- Léger et performant pour les applications temps réel
- Utilise un seul langage (JavaScript) pour frontend + backend

II Pourquoi Node.js ?

- **Asynchrone** : il utilise le non-blocking I/O, idéal pour gérer de nombreux utilisateurs en simultané.
- **Parfait pour les API REST**, microservices, sockets, etc.
- **Assistance et la formation continue en santé numérique** .
- **Écosystème riche avec NPM** (des milliers de packages gratuits)
- **Performant** pour les applications temps réel (ex : messagerie, chat, jeu en ligne)

III Les bases essentielles de Node.js

III.1 Types de données

Les types de données sont les valeurs que l'on peut manipuler dans un programme Node.js (comme dans JavaScript).

III.1.1 Types primitifs

Type	Exemple	Description
String	`"Hello"`	Chaîne de caractères
Number	`42, 3.14`	Entier ou flottant
Boolean	`true, false`	Vrai ou faux
Undefined	`let x;`	Variable déclarée mais non définie
Null	`let y = null;`	Absence volontaire de valeur
BigInt	`12345678901234567890123456n`	Très grand entier
Symbol	`Symbol("id")`	Valeur unique, souvent pour les clés d'objet

III.1.2 Types objets (complexes)

Type	Exemple	Description
Object	`nom: "Ali", age: 20`	Données clés-valeurs
Array	[1, 2, 3], ["a", "b"]	Tableau
Function	function hello() ou () =>	Fonction réutilisable
Date	new Date()	Date et heure

Exemple de code :

JS

```
let nom = "Sami";  // String
let age = 25;  // Number
let actif = true; // Boolean
let notes = [12, 15, 18];  // Array
let user = { name: "Sami", age: 25 }; // Object
```

A noter :

- typeof permet de tester le type d'une variable :

JS

```
console.log(typeof nom);   // "string
```

III.2 Syntaxe de base

La syntaxe de Node.js est identique à celle de JavaScript moderne (ES6+), mais adaptée au côté serveur.

III.3 Déclaration de variables

JS

```
let nom = "Ali"; // variable modifiable
const PI = 3.14; // constante (non modifiable)
var ancien = "Hello";  // (ancienne syntaxe, a eviter)
```

III.3.1 Conditions

JS

```
if (age >= 18) {
console.log("Majeur");
} else {
console.log("Mineur");
}
```

III.3.2 Boucles

JS

```
for (let i = 0; i < 5; i++) {
console.log(i);
}
let notes = [12, 15, 18];
notes.forEach(n => console.log(n));
```

III.4 Les opérateurs

Les opérateurs permettent de manipuler les variables et les valeurs dans ton code Node.js (comme en JavaScript).

III.4.1 Opérateurs arithmétiques

Opérateur	Nom	Exemple (a = 10, b = 3)	Résultat
+	Addition	a+b	13
-	Soustraction	a-b	7
*	Mulitplication	a*b	30
/	Division	a/b	3.33...
%	Modulo (reste)	`a % b`	1
**	Exponentiation	a**b	1000

III.4.2 Opérateurs de comparaison

Opérateur	Signification	Exemple (a = 10, b = 3)	Résultat
==	Egal (valeur)	a == "10"	True
===	Egal (valeur+ type)	a === "10"	False
!=	Différent (valeur)	a != b	True
!==	Différent (valeur + type)	a !== "10"	True
>	Supérieur à	a > b	True
<	Inférieur à	a < b	False
>=	Supérieur ou égal	a >= 10	True
<=	Inférieur ou égal	b <= 3	True

III.4.3 Opérateurs logiques

Opérateur	Signification	Exemple (a = 10, b = 3)	Résultat
&&	ET logique	`a > 5 && b < 5`	True
||	Ou logique	a === "10" || a === 10	True
!	Négation (NON)	!(a == b)	True

III.4.4 Opérateurs d'affectation

Opérateur	Exemple	Équivaut à
=	x = 5	x reçoit 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
***=**	x *= 3	x = x * 3
/=	x /= 3	x = x / 3

III.4.5 Opérateur ternaire (if rapide)

JS

```
let age = 20;
let statut = (age >= 18) ? "majeur" : "mineur";
console.log(statut); // "majeur"
```

IV Fonctions

Une fonction est un bloc de code réutilisable qui exécute une tâche spécifique. Dans Node.js (comme en JavaScript), les fonctions sont au cœur de la logique applicative.

IV.1 Déclaration de fonction (classique)

JS

```
function sayHello(name) {
  return "Hello " + name;
}
console.log(sayHello("Ali")); // Hello Ali
```

IV.2 Fonction fléchée (Arrow Function – ES6)

Syntaxe plus courte, très utilisée dans Node.js moderne.

JS

```
const add = (a, b) => {
  return a + b;
};
console.log(add(3, 4)); // 7
```

IV.3 Fonction anonyme (sans nom)

JS

```
const greet = function(name) {
  return "Hi " + name;
};
```

IV.4 Fonction dans un objet (méthode)

JS

```
const user = {
  name: "Nourhene",
  hello: function () {
    return "Bonjour " + this.name;
  }
};
console.log(user.hello()); // Bonjour Nourhene
```

IV.5 Exporter une fonction dans un module

Très important en Node.js pour structurer son projet.

Fichier : utils.js.

JS

```
function welcome(name) {
return "Welcome, " + name;
}
module.exports = welcome;
```

Fichier : app.js

JS

```
const welcome = require('./utils');
console.log(welcome("Sami")); // Welcome, Sami
```

V Créer un serveur HTTP simple

Node.js utilise require() pour importer des bibliothèques.

Exemple complet :

JS

```
const http = require('http');
http.createServer((req, res) => {
res.write("Bienvenue !");
res.end();
}).listen(3000);
```

VI Gestion des erreurs : try/catch

Le bloc try...catch permet de capturer et gérer proprement les erreurs d'exécution.

Syntaxe générale :

JS

```
try {
// Code qui pourrait causer une erreur
} catch (err) {
// Code ex cut  si une erreur survient
} finally {
// (optionnel) Toujours execute, qu il y ait erreur ou non
}
```

Exemple simple :

JS

```
try {
let x = 10;
let y = x + z; // z n est pas defini -> erreur
console.log(y);
} catch (error) {
console.log("Une erreur est survenue :", error.message);
} finally {
console.log("Bloc finally execute.");
}
```

Résultat : csharp Une erreur est survenue : z is not defined Bloc finally exécuté.

VI.1 Notes pédagogiques :

Mot-clé	Rôle
try	Contient le code potentiellement à risque
catch	Capture l'erreur si elle se produit
finally	(facultatif) Toujours exécuté à la fin

Exemple avec une fonction :

JS

```
function divide(a, b) {
try {
if (b === 0) {
throw new Error("Division par zero !");
}
return a / b;
} catch (err) {
return "Erreur : " + err.message;
}
}
console.log(divide(10, 0));  // Erreur : Division par zero
```

VI.2 Résumé

Mot-clé	Rôle
try	Contient le code potentiellement à risque
catch	Capture l'erreur si elle se produit
finally	(facultatif) Toujours exécuté à la fin

VII Importer des modules

Node.js utilise require() pour importer des bibliothèques :

JS

```
const fs = require('fs'); // module natif de fichiers
Avec ES Modules (nouveau format) :
```

VIII Middleware

Fonction exécutée avant la réponse finale. Utilisé pour :

- Authentification.
- Validation des données.
- Logs, erreurs, etc.

JS

```
app.use((req, res, next) => {
  console.log("Requete reeue !");
  next(); // continue
});
```

IX Entrée/Sortie en Node.js

IX.1 Entrée

Dans un serveur Express, les données entrantes peuvent venir de :

- **URL Parameters**

  Exemple : /api/todos/ :id

JS

```
app.get("/api/todos/:id", (req, res) => {
  console.log(req.params.id); // Entree
});
```

- **Query Parameters**

  Exemple : /api/todos ?completed=true

JS

```
app.get("/api/todos", (req, res) => {
console.log(req.query); // Entree
});
```

- **Body (JSON)**

Utilisé avec POST, PUT, etc.

JS

```
app.use(express.json()); // Important !
app.post("/api/todos", (req, res) => {
console.log(req.body); //
});
```

Ne pas oublier express.json() pour parser le body !

IX.2 Sortie

On renvoie une réponse avec res :

- **Réponse simple :** En Node.js/Express, res est l'objet qui permet d'envoyer une réponse au client (navigateur, front React, Postman...).
  - res = réponse
  - send() = envoie du texte simple

JS

```
res.send("Bienvenue !");
```

- **Réponse JSON (API REST) :** res.json() est une méthode qui permet d'envoyer un objet JavaScript en réponse, converti automatiquement en JSON (langage standard du web pour les données).

JS

```
app.get('/user', (req, res) => {
res.json({
name: "Nourhene",
age: 25 });
});
```

Résultat côté client (dans Postman ou frontend) :

JSON

```
{
"name": "Nourhene",
"age": 25
}
```

Le client reçoit des données structurées (facile à exploiter).

- **Code HTTP personnalisé :** HTTP (HyperText Transfer Protocol) est le langage que les clients et serveurs utilisent pour communiquer sur le web.

JS

```
res.status(201).json({ message: "Todo cree avec succes" });
```

Il existe plusieurs méthodes HTTP :

Méthode	Utilité
GET	Lire une ressource
POST	Créer une ressource
PUT	Modifier une ressource
DELETE	Supprimer une ressource

Exemple complet :

JS

```
app.post("/api/todos", (req, res) => {
const { title } = req.body;
if (!title) {
return res.status(400).json({ error: "Titre requis" });
}
// Simulation d enregistrement :
const newTodo = { id:1, title, completed:false };
res.status(201).json({message:"Todo ajoute", todo:newTodo});
});
```

IX.2.1 Résumé simple :

Élément	Rôle
res	Objet pour envoyer la réponse
res.json()	Réponse formatée en JSON
http	Langage de communication (GET, POST...)

X Objets

un objet est une structure qui permet de stocker des données sous forme de paires **clé : valeur.**

Ils sont très utilisés pour organiser et manipuler les données, mais aussi pour configurer des modules, gérer des requêtes, créer des réponses HTTP, etc.

JS

```
const user = {
nom: "Nourhene",
age: 22,
saluer: function() {
console.log("Bonjour !"); }};
console.log(user.nom);// acces a une propriete
user.saluer();
// appel d une methode
```

JS

```
import fs from 'fs';
```

XI Conditions dans Node.js

Les instructions conditionnelles permettent d'exécuter différents blocs de code selon que certaines conditions soient vraies ou fausses.
Node.js utilise les mêmes conditions que JavaScript.

XI.1 if...else

JS

```
let age = 20;
if (age >= 18) {
console.log("You are an adult.");
} else {
console.log("You are a minor.");
}
```

Le bloc if est exécuté si la condition est vraie. Sinon, else est exécuté.

XI.2 if...else if...else

JS

```
let score = 75;
if (score >= 90) {
console.log("Excellent");
} else if (score >= 70) {
console.log("Good");
} else {
console.log("Needs Improvement");
}
```

Permet de tester plusieurs conditions dans l'ordre.

XI.3 Opérateur ternaire (condition courte)

JS

```
let age = 16;
let status = (age >= 18) ? "Adult" : "Minor";
console.log(status); // "Minor"
```

Utilisé pour des conditions simples dans une seule ligne.

XI.3.1 switch...case

Switch est utile pour remplacer plusieurs if else quand on teste une seule variable avec plusieurs valeurs possibles.

JS

```
let day = 3;
switch (day) {
case 1:
console.log("Monday");
break;
case 2:
console.log("Tuesday");
break;
case 3:
console.log("Wednesday");
break;
default:
console.log("Unknown day");
}
```

Utilisé pour des conditions simples dans une seule ligne.

XI.4 Comparateurs dans les conditions

Comparateur	Signification	Exemple
==	Egal (valeur)	a == "5"
===	Egal (valeur + type)	a === 5
!=	Différent	a != b
!==	Différent (strict)	a !== b
> >= < <=	Comparaison numérique	x > y, etc.

XII Importer des modules

Node.js permet de modulariser son application en séparant le code en plusieurs fichiers. Pour cela, on utilise importer et exporter des fonctions, objets ou classes.

XII.1 Modules natifs (de Node.js)

Node.js propose des modules intégrés comme fs, http, path, etc.

JS

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

XII.2 Importer un fichier local (CommonJS)

Si tu as un fichier utils.js :

JS

```
// utils.js
function hello(name) {
return "Hello " + name;
}
module.exports = hello;
```

Tu l'importes ainsi dans un autre fichier :

JS

```
// app.js
const hello = require('./utils');
console.log(hello("Ali"));
```

C'est la syntaxe CommonJS, la plus utilisée dans Node.js.

XII.3 Exporter plusieurs choses

JS

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

JS

```
// app.js
const { add, sub } = require('./math');
console.log(add(5, 2)); // 7
```

XII.4 Import ES6 (avec import)

Utilisé si tu actives le mode type : "module" dans package.json

JSON

```
{
"type": "module"
}
// math.js
export function add(a, b) {
return a + b;
}
```

JSON

```
// app.js
import { add } from './math.js';
console.log(add(2, 3)); // 5
```

Avec import, les fichiers doivent avoir l'extension .js et utiliser export.

XII.5 Résumé

Syntaxe	Type de module	Exemple
require()	CommonJS (par défaut)	const x = require('./x')
import	ES Module (optionnel)	import x from './x.js'

XIII Express.js

Express.js est un framework minimaliste pour Node.js qui facilite la création d'API web.

Fonctions principales :

- Gestion des routes HTTP (GET, POST, etc.).
- Intégration de middleware (authentification, validation...).
- Création rapide d'API REST.
- Manipulation facile des requêtes et réponses.

Exemple : app.get("/users", (req, res) => res.send("Liste d'utilisateurs"););

XIV MongoDB

MongoDB est une base de données NoSQL orientée documents.

Caractéristiques :

- Structure flexible en JSON (ou BSON).
- Pas de schéma rigide → idéal pour des données qui évoluent.
- Très utilisé dans les applications modernes (ME(R)N Stack).
- Permet le scalabilité horizontale (distribuer les données).

Exemple de document :

JSON

```
{
  "nom": "Alice",
  "email": "alice@example.com",
  "age": 25
}
```

XV Mongoose

Mongoose est une bibliothèque ODM (Object Data Modeling) pour MongoDB et Node.js.

Objectif :

- Créer un modèle (schema) pour structurer les données.
- Fournit des méthodes pour CRUD (Create, Read, Update, Delete).
- Gère les relations, la validation et les hooks.

Exemple :

JS

```
const UserSchema = new mongoose.Schema({
  name: String,
  email: { type: String, required: true }
});
const User = mongoose.model("User", UserSchema);
```

XVI Concepts clés

XVI.1 API REST

Architecture qui permet de communiquer entre frontend et backend via des requêtes HTTP (GET, POST, PUT, DELETE).

XVI.2 CRUD

Opérations de base :

- Créé (POST)
- Lire (GET)
- Modifier (PUT/PATCH)
- Supprimer (DELETE)

XVI.3 Route

Chemin d'accès à une ressource via une méthode HTTP :

JS

```
app.get("/produits", function(req, res) {
  res.send("Liste des produits");
});
```

XVII JWT (JSON Web Token)

Le JWT (JSON Web Token) est un standard ouvert (RFC 7519) permettant l'échange sécurisé d'informations entre deux parties (comme un client et un serveur), sous forme d'un jeton encodé en base64. Le JWT (JSON Web Token) est un standard ouvert (RFC 7519) permettant l'échange sécurisé d'informations entre deux parties (comme un client et un serveur), sous forme d'un jeton encodé en base64.

XVII.1 Objectif

Authentifier les utilisateurs sans stocker de session côté serveur. Il permet au frontend de prouver qu'il est connecté sans renvoyer les identifiants à chaque requête.

XVII.2 Fonctionnement

1. L'utilisateur se connecte (login).
2. Le serveur génère un JWT signé avec un secret.

3. Ce jeton est renvoyé au client.
4. À chaque requête suivante, le client envoie le JWT dans les headers (Authorization).
5. Le serveur vérifie la signature du JWT pour authentifier.

XVII.3 Structure d'un JWT

Un JWT contient 3 parties séparées par des points (.).

- **Header :** Algo de signature (ex : HS256).
- **Payload :** Données (id utilisateur, rôle...).
- **Signature :** HMAC SHA256 + secret.

XVII.4 Avantages

Avantage	Description
Sécurisé	Signé avec un secret connu du serveur
Sans session	Pas besoin de stockage serveur
Rapide	Moins de requêtes et stockage local (localStorage/cookie)
Statless	Facile à utiliser avec les APIs REST

XVII.5 Limites / Risques

- Ne pas stocker dans localStorage si l'app est exposée au XSS.
- Jeton trop long à expirer peut être dangereux.
- Si le JWT_SECRET est compromis $\rightarrow$ toute sécurité est perdue.

XVIII bcrypt

XVIII.1 Définition

bcrypt est une fonction de hachage de mots de passe conçue pour stocker les mots de passe de manière sécurisée dans une base de données. Contrairement à un simple encodage, le hachage est irréversible.

XVIII.2 Objectif

Empêcher un attaquant de récupérer le mot de passe d'un utilisateur, même s'il accède à la base de données.

XVIII.3 Fonctionnement

1. L'utilisateur s'inscrit avec un mot de passe.
2. Le serveur hash ce mot de passe avec bcrypt + un sel (salt) aléatoire.
3. Seul le hash est stocké en base.
4. Lors de la connexion, bcrypt.compare() permet de vérifier si le mot de passe fourni correspond au hash.

XVIII.4 Exemples

JS

```
const bcrypt = require("bcryptjs");
const password = "monMotDePasse123";
const hashed = await bcrypt.hash(password, 10);
// 10 = salt rounds
```

En base, on stocke :

JS

```
$2a$10$6tKXU21l7MbGTxhCfsQtxeWvzy8VPQDTSnCwz4T8mRpEK0qB/Nz7e
```

XVIII.5 Avantages

Avantage	Description
Sel intégré	Rend chaque hash unique, même pour un même mot de passe
Lent volontairement	Rend les attaques par brute-force très coûteuses
Facile à intégrer	Utilisé dans presque tous les systèmes Node.js
Sécurisé	Résistant aux attaques classiques (rainbow tables, etc.)

XVIII.6 Conseils

- Toujours utiliser un salt (intégré dans bcrypt).
- Ne jamais stocker les mots de passe en clair.
- Ne pas diminuer le nombre de saltRounds (10 ou plus recommandé).

XVIII.7 Schéma visuel à insérer (proposition)

Inscription :

```yaml
Mot de passe utilisateur -> [bcrypt.hash + salt]
-> Hash en base de donn es
```

Connexion :

```JS
Mot de passe saisi + Hash stock  -> [bcrypt.compare] ->
true / false
```

Chapitre II

Projet guidé : To-Do List API

I Objectif pédagogique

Mettre en œuvre les compétences acquises en Node.js, Express, MongoDB et Mongoose à travers un projet concret : la création d'une API REST pour une application de gestion de tâches (To-Do List) avec authentification.

II Étapes du projet

II.1 Structure du projet

```
todo-api
 models/          → Modèles Mongoose (User, Todo)
 controllers/     → Logique métier (authController, todoController)
 routes/          → Routes Express
 middleware/      → Middleware JWT d'authentification
 .env             → Variables d'environnement
 index.js         → Point d'entrée de l'API
```

III Étapes de développement

III.1 Initialisation du projet

1. npm init -y

2. npm install express mongoose dotenv cors bcryptjs jsonwebtoken

III.2 Configuration de l'environnement

- **Fichier `.env` :**

JS

```
MONGO_URI=mongodb+srv://<username<password>@cluster.mongodb.
net/todolist
JWT_SECRET=motdepasseultrasecret
PORT=5000
```

III.3 Point d'entrée (index.js)

JS

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
require("dotenv").config();
const todoRoutes = require("./routes/todoRoutes");
const authRoutes = require("./routes/authRoutes");
const app = express();
app.use(cors());
app.use(express.json());
mongoose.connect(process.env.MONGO_URI,
{useNewUrlParser: true, useUnifiedTopology:true})
.then(() => console.log("MongoDB connecte"))
.catch(err => console.error("Erreur MongoDB:", err));
app.use("/api/todos", todoRoutes);
app.use("/api/auth", authRoutes);
const PORT = process.env.PORT || 5000;
app.listen(PORT, () =>
console.log("Serveur demarre sur le port", PORT));
```

III.4 Modules principaux

III.4.1 Modèle Todo (models/Todo.js)

JS

```
const mongoose = require("mongoose");
const todoSchema = new mongoose.Schema({
title: { type: String, required: true },
completed: { type: Boolean, default: false },
userId: {type: mongoose.Schema.Types.ObjectId, ref: "User"}
});
module.exports = mongoose.model("Todo", todoSchema);
```

III.4.2 Modèle Utilisateur (models/User.js)

JS

```
const mongoose = require("mongoose");
const bcrypt = require("bcrypt");
const UserSchema = new mongoose.Schema({
name: String,
email: { type: String, required: true, unique: true },
password: { type: String, required: true}});
UserSchema.pre("save", async function (next) {
if (!this.isModified("password")) return next();
this.password = await bcrypt.hash(this.password, 10);
next();});
module.exports = mongoose.model("User", UserSchema);
```

III.4.3 Authentification (JWT + bcrypt)

- **Route Register**

1. Hash du mot de passe avec bcrypt.

2. POST /api/register.

- **Route Login**

1. POST /api/login.
2. Génération d'un token JWT avec jwt.sign.

- **Middleware auth.js**

1. Permet de protéger les routes privées :

JS

```
const jwt = require("jsonwebtoken");
module.exports = function (req, res, next) {
const authHeader = req.headers.authorization;
if (!authHeader) return res.status(401).json(
{ message: "Token manquant" });
const token = authHeader.split(" ")[1];
try {
const decoded = jwt.verify(token, process.env.
JWT_SECRET);
req.userId = decoded.id;
next();
} catch {
res.status(401).json({ message:"Token invalide"});}};
```

III.5 Contrôleurs

- **authController.js :**

JS

```
const User = require("../models/User");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcrypt");
exports.register = async (req, res) => {
const { name, email, password } = req.body;
try {
const user = await User.create({name,email,password});
res.status(201).json({ message: "Utilisateur cr    ",
user });
} catch (err) {
res.status(400).json({ error: "Email d j   utilis
ou invalide." });
}
};
exports.login = async (req, res) => {
const { email, password } = req.body;
try {
const user = await User.findOne({ email });
if (!user) return res.status(401).json({ error:
"Utilisateur introuvable" });
const isMatch = await bcrypt.compare(password,
user.password);
if (!isMatch) return res.status(401).json({ error:
"Mot de passe incorrect" });
const token = jwt.sign({ id: user._id }, process.env.
JWT_SECRET, {
expiresIn: "1d"
});
res.json({ message: "Connexion r ussie", token });
} catch (err) {
res.status(500).json({ error: "Erreur serveur" });}};
```

- **todoController.js**

JS

```
const Todo = require("../models/Todo");
exports.getTodos = async (req, res) => {
const todos = await Todo.find({ userId: req.userId });
res.json(todos);};
exports.createTodo = async (req, res) => {
const { title } = req.body;
const todo = new Todo({ title, userId: req.userId });
await todo.save();
res.status(201).json(todo);};
exports.updateTodo = async (req, res) => {
try {
const { id } = req.params;
const todo = await Todo.findByIdAndUpdate(id,
req.body, { new: true });
if (!todo) {
return res.status(404).json(
{message:"Tache non trouvee"});}
res.json({ message: "Tache mise a jour
avec succes", todo });
}catch (error) {
res.status(500).json({ message: "Erreur serveur" });}};
exports.deleteTodo = async (req, res) => {
await Todo.findOneAndDelete({ _id: req.params.id,
userId: req.userId });
res.json({ message: "Tache supprimee" });
};
```

III.6 Routes Express

- **authRoutes.js**

JS

```
const express = require("express");
const router = express.Router();
const { register, login } = require("../controllers/
authController");
router.post("/register", register);
router.post("/login", login);
module.exports = router;
```

- **todoRoutes.js**

JS

```
const express = require("express");
const router = express.Router();
const {
getTodos, createTodo, updateTodo, deleteTodo
} = require("../controllers/todoController");
const auth = require("../middleware/auth");
router.use(auth);
router.get("/", getTodos);
router.post("/", createTodo);
router.put("/:id", updateTodo);
router.delete("/:id", deleteTodo);
module.exports = router;
```

IV Test de l'API

Nous pouvons testé avec postman ou arc.

Action	Méthode	Endpoint	Corps requis (JSON)	Auth
Register	`POST`	/api/register	`{"username": "nom", "email": "...", "password": "..." }`	Non
Login	`POST`	/api/login	`{"email": "...", "password": "..." }`	Non
Créer tâche	`POST`	/api/todos	`{"title": "ma tâche"}`	Oui (Bearer)
Voir les tâches	`GET`	/api/todos	–	Oui (Bearer)
Modifier tâche	`PUT`	/api/todos/ :id	`{"title": "...", "completed": true}`	Oui (Bearer)
Supprimer tâche	`DELETE`	/api/todos/ :id	–	Oui (Bearer)

V Déploiement de l'API Node.js

Pour le déploiement nous allons utilisé la plateforme **Render.com**.

Ces avantages sont :

- Simple.
- Gratuite.
- Idéale pour les débutants.

VI Objectif

Déployer notre API (Node.js + Express + MongoDB) sur un serveur distant accessible via une URL publique.

VII Étapes complètes

1. **Pousser le projet sur GitHub :**
   - Crée un dépôt GitHub (ex : todo-api-node).
   - Initialise Git dans ton projet :

— git init

— git add .

— git commit -m "Initial commit"

— git remote add origin https ://github.com/ton-user/ton-projet.git

— git push -u origin main

2. **Créer un service sur Render :**
   - Connecte-toi sur Render.com
   - Clique sur "New" → "Web Service"
   - Connecte ton compte GitHub
   - Sélectionne ton repo todo-api-node
3. **Configurer le service :**
   - **Nom :** todo-api (ou autre)
   - **Environment :** Node
   - **Build Command :** npm install
   - **Start Command :** node index.js (ou npm start si tu as défini dans package.json)
   - **Root Directory :** (laisser vide si ton code est à la racine)
   - **Region :** Frankfurt (Europe) ou selon ton choix.
4. **Ajouter les variables d'environnement :**
   - Dans l'onglet Environment → Add Environment Variable :

Variable	Valeur
PORT	10000 (Render attribue automatiquement le bon port)
MONGO_URI	(Lien MongoDB Atlas, exemple : mongodb+srv ://...)
JWT_SECRET	motsecret123 (ou n'importe quel mot secret)

Note : Ne pas fixer PORT=5000. Render fournit sa propre variable d'environnement PORT, donc :

```JS
const PORT = process.env.PORT || 5000;
```

5. **Déploiement automatique :**

Une fois connecté à GitHub :

- Chaque push sur GitHub redéclenchera le déploiement automatiquement.

- Une URL publique sera générée. Exemple : https ://todo-api-node.onrender.com

6. **Tester l'API déployée (avec ARC ou Postman) :**
   - Ajoute l'URL dans ton client HTTP :
     — POST https ://todo-api-node.onrender.com/api/register
     — GET https ://todo-api-node.onrender.com/api/todos
     — Ajouter Authorization : Bearer <token> dans l'en-tête pour les routes protégées.

Bonnes pratiques Render :

- Ne jamais publier le .env dans ton repo (ajoute .env dans .gitignore).
- Organise ton projet : routes/, controllers/, models/, middleware/.
- Pense à sécuriser tes routes avec JWT + bcrypt (déjà fait dans notre projet !).

Chapitre III

Partie Frontend

I Introduction à React.js

React.js est une bibliothèque JavaScript développée par Meta pour construire des interfaces utilisateur modernes et dynamiques.

II Pourquoi React ?

- Création d'interfaces réactives avec peu de code.
- Utilisation des composants réutilisables.
- Très rapide grâce au Virtual DOM.
- Idéal pour construire des Single Page Applications (SPA).

III Concepts fondamentaux de React

III.1 Composant Fonctionnel

Un composant fonctionnel est une fonction JavaScript qui retourne du JSX (HTML + JS combiné).

JSX

```
function Welcome() {
return <h1>Bienvenue !</h1>;
}
```

III.2 JSX (JavaScript XML)

JSX est une extension de syntaxe qui permet d'écrire du code HTML à l'intérieur de JavaScript.

JSX

```
const element = <h1>Hello, world!</h1>;
```

III.3 useState

Ce hook permet de gérer l'état local dans un composant.

JSX

```
import { useState } from "react";
const [count, setCount] = useState(0);
```

III.4 useEffect

Permet d'exécuter une fonction après le rendu du composant. Idéal pour les appels API.

JSX

```
useEffect(() => {
fetchData();
}, []);
```

IV Organisation d'un projet React

```
src/
 assets/
    logo_to_do_sans_bg.png
 components/
    login.jsx
    register.jsx
    ToDoList.jsx
    navbar.jsx
    PrivateRoute.jsx
    todo.css
    navbar.css
    login.css
 services/
    api.js
 App.jsx
 index.js
```

V Connexion au Backend avec Axios

Axios est une bibliothèque JavaScript qui permet de faire des requêtes HTTP (GET, POST, PUT, DELETE...) vers un serveur ou une API, depuis :

- Une application frontend (React, Vue, etc.)
- Un projet Node.js

V.1 Pourquoi utiliser Axios ?

Axios te permet de communiquer avec un backend (comme ton API Express/MongoDB) pour :

- Récupérer des données (GET).
- Envoyer des données (POST).
- Modifier des données (PUT).

- Supprimer des données (DELETE).

V.2 Exemple

JSX

```
import axios from 'axios';
const API = axios.create({
baseURL: 'https://todo-api-tkf8.onrender.com/api',
});
API.interceptors.request.use((req) => {
const token = localStorage.getItem('token');
if (token) req.headers.Authorization = `Bearer ${token}`;
return req;
});
export default API;
```

V.3 Avantages d'Axios

Avantage	Détail
Support natif de JSON	Pas besoin de .json() comme avec fetch
Gestion automatique des promesses	.then() / .catch()
Support des délais / timeouts	Tu peux couper une requête si elle est trop longue
Envoie facile des headers (ex : token JWT)	axios.get(url, headers : Authorization : "Bearer ...")
Upload de fichiers (form-data)	Facile à faire avec axios.post()

VI Authentification avec JWT

- Le token JWT est stocké dans localStorage.
- Lors de chaque appel API, il est ajouté dans l'en-tête Authorization.
- Pour se déconnecter :

```JSX
localStorage.removeItem("token");
```

VII Fonctionnalités de la To-Do List

- Ajouter une tâche :

```JSX
await API.post("/todos", { title });
```

- Supprimer une tâche :

```JSX
await API.delete(`/todos/${id}`);
```

- Modifier une tâche :

```JSX
await API.put(`/todos/${id}`, { title });
```

- Marquer comme complétée :

```JSX
await API.put(`/todos/${id}`, { completed: true });
```

VIII Protection des Routes

Si le token est manquant → redirection vers la page de login :

JSX

```
if (!localStorage.getItem("token")) {
navigate("/login");
}
```

IX CSS pour chaque page

Des fichiers .css dédiés sont associés à chaque composant :

- login.css
- register.css
- todo.css
- navbar.css

X Exemple de composant

X.1 ToDoList.jsx

JSX

```
return (
<li>
{editId === todo._id ? (
<>
<input value={editTitle} onChange={
(e) => setEditTitle(e.target.
value)} />
<button onClick={() => saveEdit(todo._id)}>
</>
) : (
<>
</button>
<span style={{ textDecoration:
todo.completed ? "line-through" : "none" }}>
{todo.title}
</span>
<button onClick={() => startEdit(todo)}>
</button>
<button onClick={() => toggleCompleted(todo)}>
{todo.completed ? " " : "  "}
</button>
</>
)}
<button onClick={() => deleteTodo(todo._id)}>    </button>
</li>
);
```

XI Déploiement du frontend React

Pour le déploiement nous allons utilisé la plateforme **Render.com**.

Ces avantages sont :

- Simple.
- Gratuite.
- Idéale pour les débutants.

XII Objectif

Déployer notre frontend (React) sur un serveur distant accessible via une URL publique.

XIII Étapes complètes

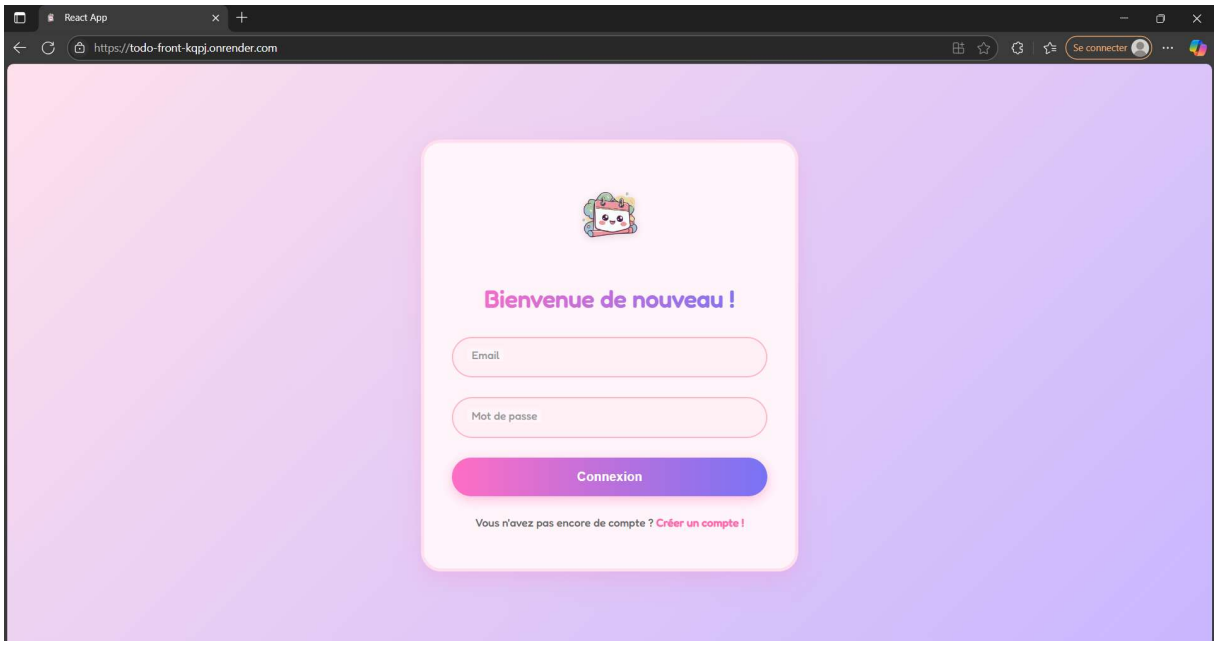
1. **Pousser le projet sur GitHub :**
   - Crée un dépôt GitHub (ex : todo-front).
   - Initialise Git dans ton projet :
     - git init
     - git add .
     - git commit -m "Initial commit"
     - git remote add origin https ://github.com/ton-user/ton-projet.git
     - git push -u origin main
2. **Créer un service sur Render :**
   - Connecte-toi sur Render.com
   - Clique sur "New" → "Web Service"
   - Connecte ton compte GitHub
   - Sélectionne ton repo todo-front
3. **Configurer le service :**
   - **Nom :** todo-front (ou autre)
   - **Build Command :** npm run build
4. **Déploiement automatique :**

   Une fois connecté à GitHub :

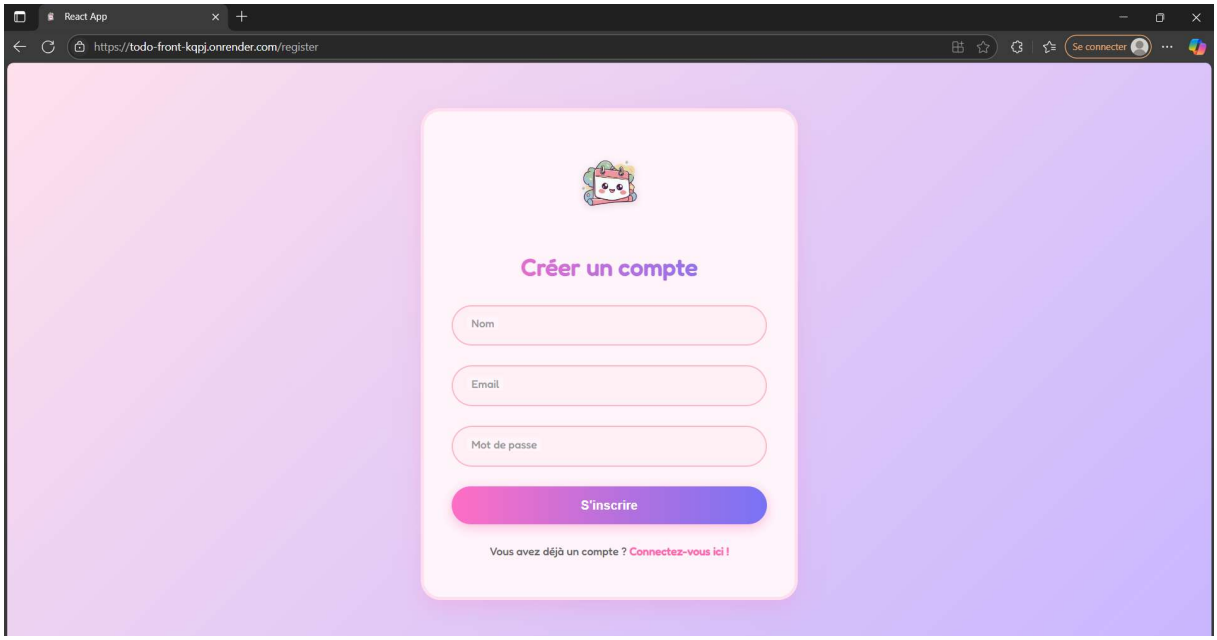
- Chaque push sur GitHub redéclenchera le déploiement automatiquement.
- Une URL publique sera générée. Exemple : https ://todo-front-kqpj.onrender.com

XIV Réalisation

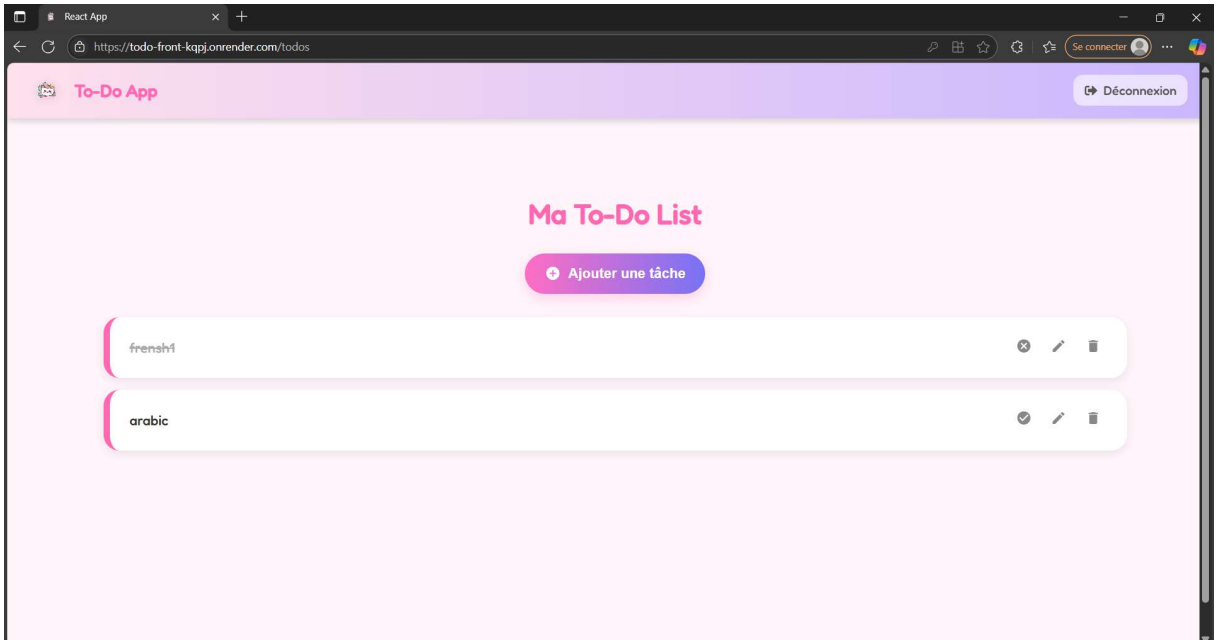
- Interface Login



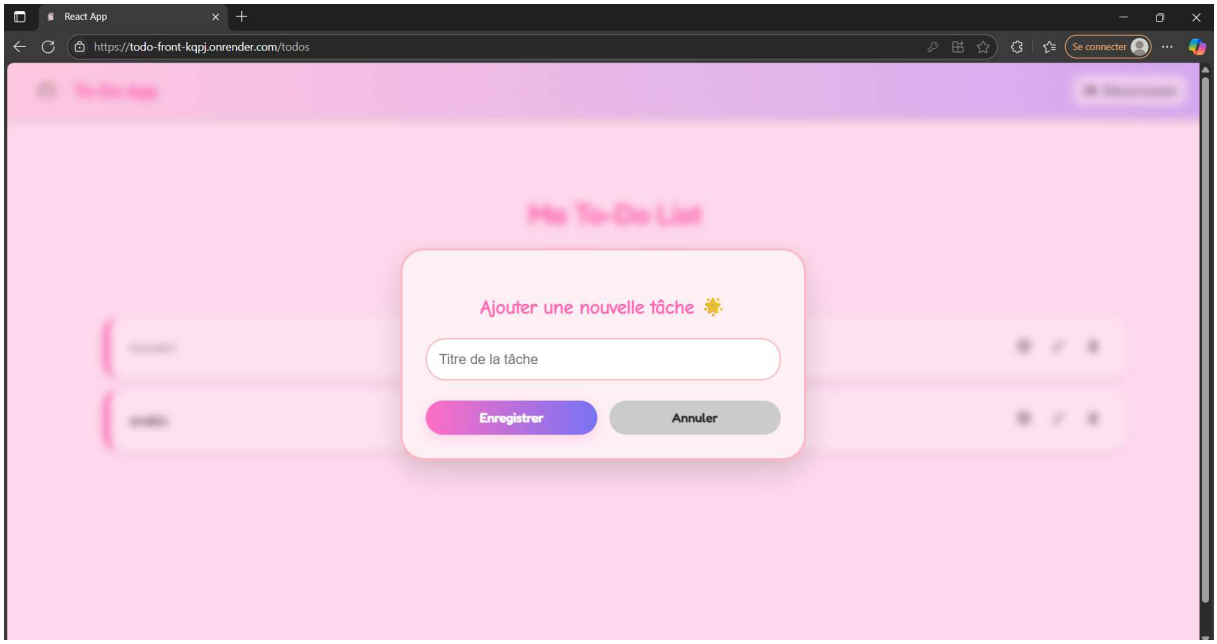
- Interface Inscription



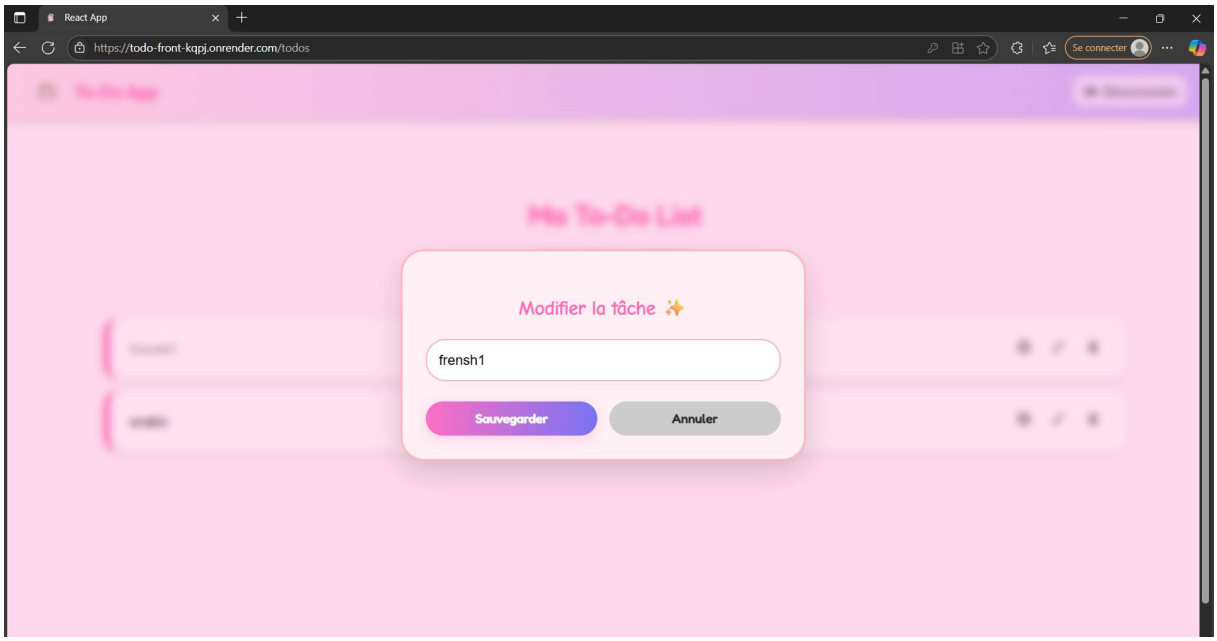
- Interface ToDo



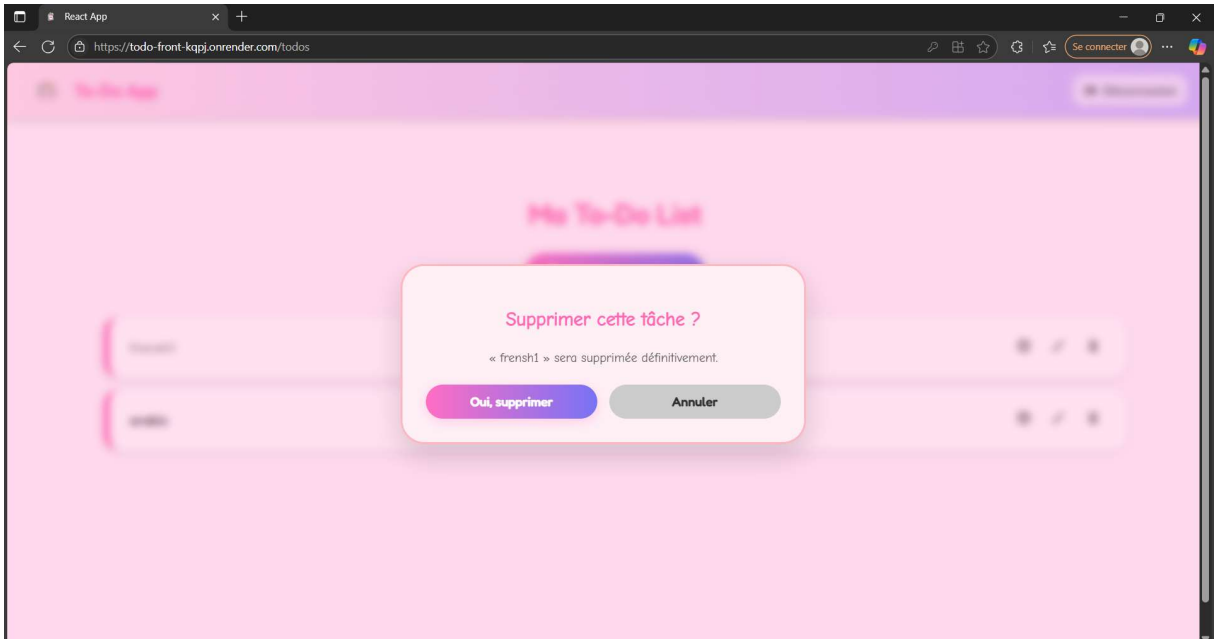
- Interface Ajouter tâche



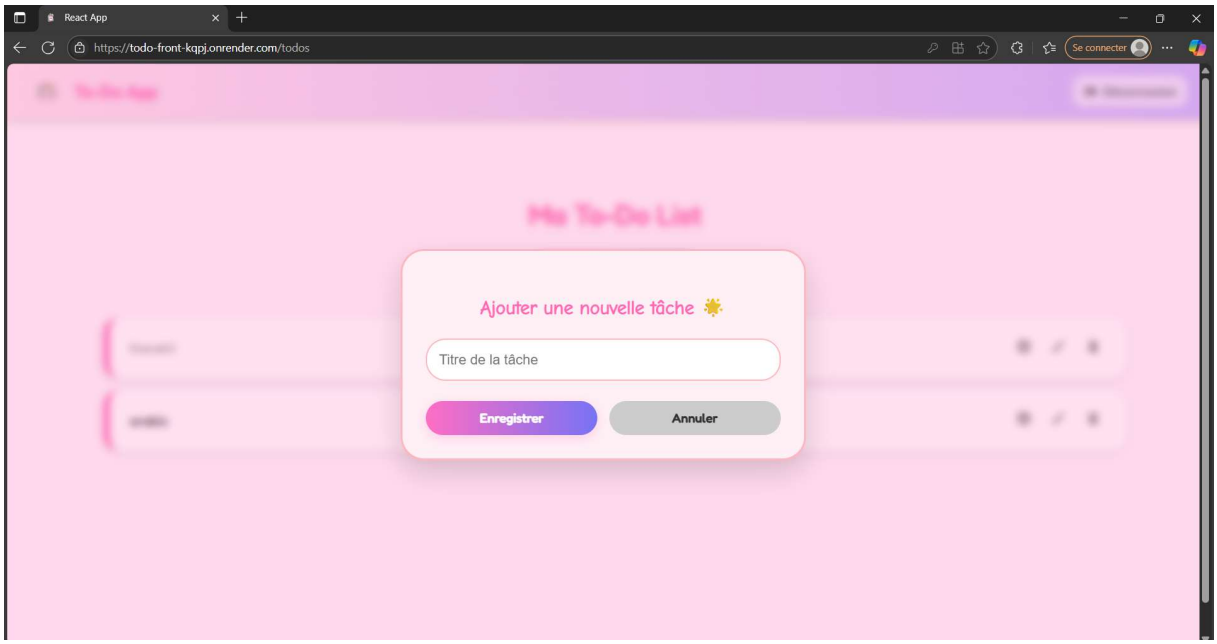
- Interface Modifier tâche



- Interface Supprimer tâche



- Interface Ajouter tâche


React App
https://todo-front-kqpj.onrender.com/todos
Se connecter
Ajouter une nouvelle tâche
Titre de la tâche
Enregistrer
Annuler

Conclusion Générale

Au terme de ce travail, nous avons pu acquérir une compréhension solide des fondamentaux du développement backend avec Node.js et Express.js, en passant par la gestion de base de données avec MongoDB via Mongoose, jusqu'à la mise en place d'une API REST sécurisée par des techniques d'authentification comme JWT et bcrypt.

Le développement de l'application To-Do List a permis de mettre en pratique ces connaissances dans un projet complet et structuré, depuis la conception du serveur jusqu'à l'intégration avec un frontend développé en React.js. Cette expérience a également permis de découvrir l'importance de l'interaction entre les différentes couches d'une application web moderne, ainsi que les bonnes pratiques en matière de sécurité et d'organisation du code.

Cette initiation au développement fullstack ouvre la voie vers des projets plus complexes, tout en constituant une base solide pour approfondir davantage les technologies JavaScript et les architectures web actuelles.

Annexes

Ce guide repose sur une stack de développement backend moderne basée sur Node.js, Express.js et MongoDB. Voici les étapes pour installer les outils nécessaires.

- **Installer Node.js & npm :**
  1. Méthode recommandée (via NodeSource) : sudo apt install -y nodejs
  2. Vérifier l'installation : npm -v
- **Initialiser un projet Node.js :**
  1. mkdir backend-api
  2. cd backend-api
  3. npm init -y
- **Installer Express.js :**
  1. npm install express
- **Installer MongoDB localement :**
  1. sudo apt update
  2. sudo apt install -y mongodb
  3. sudo systemctl start mongodb
  4. sudo systemctl enable mongodb
- **Installer Mongoose (ODM pour MongoDB) :**
  1. npm install mongoose
- **Installer et utiliser Postman (pour tester l'API) :**
  1. Télécharger Postman Desktop : https ://www.postman.com/downloads/
  2. Ou utiliser la version Web : https ://web.postman.com

- **Lancer le serveur avec nodemon (si installé) :**
  1. npm nodemon index.js
- **Créer un projet React :**
  1. mkdir todo-frontend
  2. cd todo-frontend
  3. npx create-react-app .
- **Installer Axios (consommation d'API REST) :**
  1. npm install axios
- **Lancer le frontend :**
  1. npm start