Cours tutoriel sur la technologie Next.js

ADOM Tadjoudine, DJODI Rachad

April 1, 2025

Contents

1 Introduction

1.1 Présentation de Next.js

Next.js est un framework React qui fournit les éléments de base pour créer des applications web full-stack.

On utilise React pour créer l'interface utilisateur, et Next.js pour des fonctionnalités ou optimisations supplémentaires.

Sous le capot, Next.js configure tous les outils nécessaires à React, ce qui nous permet de nous concentrer sur la création de l'application plutôt que de passer du temps sur les configurations.

1.2 Caractéristiques principales

Les principales fonctionnalités de Next.js sont :

- **Routage** : Routage basé sur le système de fichiers.
- **Rendu** : Rendu côté **client** et **serveur** avec les composants clients et serveurs.
- **Récupération des données** : Récupération des données simplifiée avec async/await dans les composants et une API ***fetch*** pour la mémorisation des requêtes, la mise en cache, la revalidation.
- **Style** : Prise en charge des différentes méthodes de styles comme **CSS Module**, **Tailwind CSS**, **CSS-in-JS**, etc.
- **API Routes** : Création des endpoints API, qui constituent le backend de l'application, pour bien séparer l'interface utilisateur.
- **Optimisation** : Optimisation des liens, des images, des polices, etc., pour améliorer certains éléments web et l'expérience utilisateur.

2 Pré-requis et installation de Next.js

2.1 Pré-requis

Pour apprendre efficacement Next.js, il est utile de se familiariser avec **JavaScript**, **React.js** et les concepts de développement web associés (**HTML**, **CSS**). Nous devons installer une version **20** de **Node.js** ou supérieure. On peut utiliser un éditeur de texte de choix comme **VSCode** (conseillé).

2.2 Installation de Next.js

Pour créer un nouveau projet Next.js, il est recommandé d'utiliser la commande suivante :

```
npx create-next-app@latest my-app
```

Lors de l'installation, nous aurons les invites suivantes :

```
Would you like to use TypeScript? No / Yes
Would you like to use ESLint? No / Yes
Would you like to use Tailwind CSS? No / Yes
Would you like your code inside a `src/` directory? No / Yes
Would you like to use App Router? (recommended) No / Yes
Would you like to use Turbopack for `next dev`? No / Yes
Would you like to customize the import alias (`@/*` by default)? No / Yes
What import alias would you like configured? @/*
```

- **npx** : Outil de npm pour exécuter les paquets.

- **create-next-app** : Interface en ligne de commande qui configure automatiquement tout le projet Next.js.
- **@latest** : Pour spécifier d'utiliser la dernière version de Next.js.

3 Structure du projet

Lorsqu'on accède à la racine du projet, on a cette structure :

- **/app** : Nouveau système de routage avec App Router. C'est le dossier principal du projet. Il contient :
  - **/layout.tsx** : Définit la structure de mise en page de l'application.
  - **/page.tsx** : Page d'accueil du projet.
- **/public** : Contient les assets statiques (images, icônes, etc.).
- **/styles** : Contient les fichiers de style globaux.
  - **globals.css** : Fichier CSS global.
  - **tailwind.css** : Configuration pour Tailwind CSS (si utilisé).
- **package.json** : Contient la liste des dépendances du projet.
- **.env** : Contient les variables d'environnement.
- **.eslintrc.json** : Fichier de configuration pour ESLint.
- **jsconfig.json** : Fichier de configuration pour JavaScript.
- **tsconfig.json** : Fichier de configuration pour TypeScript.
- **next-env.d.ts** : Fichier de déclaration TypeScript pour Next.js.
- **next.config.js** : Fichier de configuration de Next.js.
- **tailwind.config.js** : Fichier de configuration pour Tailwind CSS.
- **postcss.config.mjs** : Fichier de configuration pour PostCSS.

4 App Router (Système de Routage et Pages)

4.1 Système de routage

Le système de routage, appelé **App Router**, est basé sur la structure du dossier **app**. Cela signifie que l'organisation des dossiers à l'intérieur du répertoire **app** définit automatiquement les routes de l'application.

Chaque dossier situé dans **app**, contenant un fichier **page.tsx** ou **page.js**, représente une route. Le fichier **page.tsx** ou **page.js** correspond à la page associée à cette route.

Il existe différents types de routes dans une application Next.js mais nous allons étudier.

- **Routes statiques**
- **Routes statiques imbriquées**
- **Routes dynamiques**
- **Routes groupées**

4.2 Routes statiques

En Next.js, les routes statiques sont des pages définies à l'avance et servies telles quelles, sans requête côté serveur ni rendu dynamique.

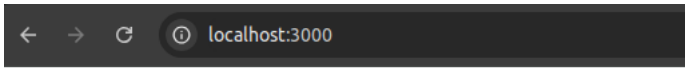
Nous allons créer une route statique pour la page d'accueil de l'application. Pour cela, nous ajoutons le code ci-dessous dans le fichier **app/page.tsx**.

```
export default function Home() {
    return (
        <div>
            <h1>Bienvenue sur le cours de Next.js 15</h1>
            <p>Page d'accueil de Next.js 15</p>
        </div>
    )
}
```

Cette page est le point d'entrée de l'application. La route de cette page est : /

Pour accéder à la page, on démarre le serveur : **npm run dev**, puis on tape l'URL : **http://localhost:3000/**

Voici à quoi doit ressembler la page :



Bienvenue sur le cours de nextjs 15

page d'accueil de nextjs 15

4.3 Création d'une route statique imbriquée

Les routes statiques imbriquées en Next.js sont des routes organisées en sous-dossiers, ce qui permet de structurer proprement l'application.

Nous allons continuer en créant une route statique imbriquée.

Dans le fichier **app/about/page.jsx** :

```
export default function AboutPage(){
    return (
        <div>
            <h1>À propos de nous</h1>
            <p>Ceci est la page à propos de nous</p>
        </div>
    )
}
```

Pour cette page, la route est : **/about**.

Voici à quoi doit ressembler la page :

C'est ainsi qu'on crée les routes en suivant les chemins des dossiers.

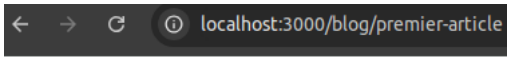
4.4 Création d'une route dynamique

Les routes dynamiques permettent de créer des routes avec des paramètres variables.
On crée une route dynamique en utilisant les **crochets** []. Par exemple, pour afficher les articles d'un blog avec un slug, on utilise le fichier **app/blog/[slug]/page.tsx** :

```
export default async function BlogPost({ params }: { params: { slug: string } }) {
    const { slug } = params;
    return (
        <div>
            <h1>L'article { slug }</h1>
            <p>Ceci est l'article avec le slug { slug }</p>
        </div>
    )
}
```

L'objet **params** est un prop qui contient les URLs dynamiques.

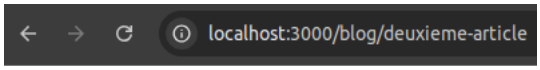
Exemple de route : **/blog/premier-article**



L'article premier-article

ceci est l'article avec le slug premier-article

/blog/deuxieme-article



L'article deuxieme-article

ceci est l'article avec le slug deuxieme-article

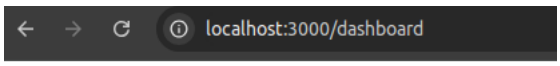
4.5 Route groupée

Les routes groupées permettent de structurer le code sans affecter les URLs. On les crée en utilisant les parenthèses **( )**.

Créons un exemple de route groupée. Dans le fichier **app/(admin)/dashboard/page.tsx** :

```
export default function DashboardPage(){
    return (
        <div>
            <h1>Ceci est la page de dashboard</h1>
        </div>
    )
}
```

La route est : **/dashboard**

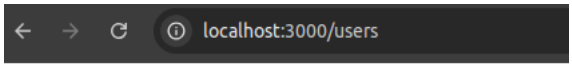


ceci est la page de dashboard

Exemple 2 : dans le fichier **app/(admin)/users/page.tsx** :

```
export default function UsersPage() {
    return (
        <div>
            <h1>Bienvenue sur la page des utilisateurs</h1>
        </div>
    )
}
```

La route est : **/users**



Bienvenue sur la page de users

Le dossier **(admin)** est ignoré dans l'URL, c'est une façon d'organiser le code.

4.6 Création d'une route de redirection

Les routes de redirection permettent de rediriger une route vers une autre route.

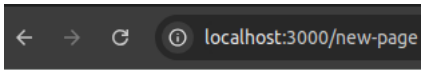
Dans le fichier : **app/old-page/page.tsx**

```
import { redirect } from 'next/navigation';

export default function OldPage() {
  redirect('/new-page');
}
```

Dans le fichier : **app/new-page/page.tsx**

```
export default function NewPage() {
  return (
    <div>
      <h1>Nouvelle page</h1>
      <p>Tu as été redirigé ici</p>
    </div>
  );
}
```

Lorsqu'on accède à **/old-page**, on sera redirigé vers : **/new-page**



Nouvelle page

Tu as été redirigé ici

4.7 Les fichiers de routage

Plusieurs fichiers de routage sont reconnus par Next.js avec différents rôles.

- **not-found.tsx** : Personnaliser une page de 404.

  Dans le fichier **app/not-found.tsx** :

```tsx
export default function NotFound() {
  return (
    <div>
      <h1>404 - Page non trouvée</h1>
      <p>Désolé, la page que vous cherchez n'existe pas.</p>
    </div>
  );
}
```

Nous allons essayer d'accéder à une route qui n'existe pas : **/papa**

- **layout.tsx** : Permet de partager une structure commune entre la page principale et les pages des sous-dossiers d'un répertoire.

  Dans le fichier **app/(admin)/dashboard/layout.tsx** :

```tsx
export default function DashLayout({ children }: { children: React.ReactNode }) {
    return (
        <div>
        <header>
          <h1>Mon Site</h1>
        </header>
        <main>{children}</main>
        <footer>
          <p>© 2025 Mon Site</p>
        </footer>
        </div>
    )
}
```

Dans le fichier **app/(admin)/dashboard/page.tsx** :

```
export default function DashboardPage(){
    return (
        <div>
            <h1>Ceci est la page de dashboard</h1>
        </div>
    )
}
```

Dans le fichier : **app/(admin)/dashboard/settings/page.tsx** :

```
export default function SettingsPage() {
    return (
        <div>
            <h1>Les paramètres</h1>
        </div>
    )
}
```

Si on accède à ces deux routes : **/dashboard** ou **/dashboard/settings**, les deux pages ont une structure commune.

- **loading.tsx** : Permet d'afficher une page avec un état de chargement.

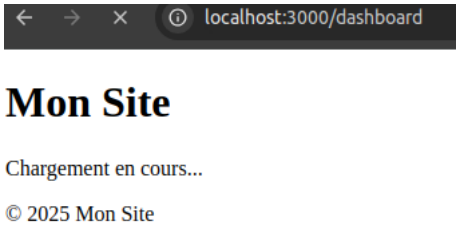
Dans le fichier : **app/(admin)/dashboard/loading.tsx** :

```
export default function Loading() {
    return (
      <div>
        <p>Chargement en cours...</p>
      </div>
    );
}
```

Dans le fichier : **app/(admin)/dashboard/page.tsx** :

```
export default async function DashboardPage(){
    await new Promise((resolve) => setTimeout(resolve, 8000));
    return (
        <div>
            <h1>Ceci est la page de dashboard</h1>
        </div>
    )
}
```

Si on accède à **/dashboard**, le contenu de **loading.tsx** sera affiché pendant 8 secondes.



- **error.tsx** : Permet d'afficher une page d'erreur lorsqu'une page quelconque génère une erreur.

  Dans le fichier **app/products/error.tsx** :

```
'use client';

export default function Error({
  error,
  reset,
}: {
  error: Error;
  reset: () => void;
}) {
  return (
    <div>
      <h2>Une erreur s'est produite</h2>
      <p>{error.message}</p>
      <button onClick={reset}>Réessayer</button>
    </div>
  );
}
```

Ici, nous avons ajouté la directive **'use client';** pour indiquer que ce composant sera exécuté côté client (dans le navigateur).

Dans le fichier **app/products/page.tsx** :

```
export default async function Products() {
    throw new Error('Erreur lors du chargement des produits');
}
```

Lorsqu'on accède à la route : **/products**

5 Navigation avec le composant Link

Le composant **Link** permet de naviguer entre deux pages sans recharger la page, ce qui signifie que la transition entre les deux pages se fait dans le navigateur par **JavaScript**.

Dans **app/page.tsx** :

```
import Link from "next/link"
export default function Page() {
    return (
        <div>
        <h1>Bienvenue sur Next.js 15 !</h1>
        <p>Ceci est la page d'accueil.</p>
        <nav>
          <Link href="/about">À propos</Link> | |
          <Link href="/">Accueil</Link> | |
          <Link href="/dashboard">Tableau de bord</Link>
        </nav>
      </div>
    )
}
```

Lorsqu'on accède à la page d'accueil, on doit voir les liens :

6 Navigation Programmée avec useRouter()

Permet de naviguer entre les pages en utilisant, par exemple, un bouton plutôt qu'un lien.

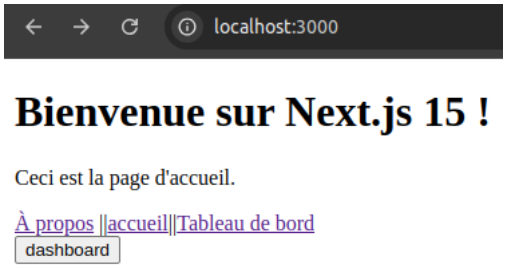
Dans le fichier : **app/components/RedirectButton.tsx**, on va créer un composant réutilisable :

```
"use client"

import { useRouter } from "next/navigation"

export default function RedirectButton() {
    const route = useRouter()
    return (
        <button onClick={() => route.push("/dashboard")}>
            Dashboard
        </button>
    )
}
```

On ajoute ce bouton à la page d'accueil dans le fichier : **app/page.tsx**

```
import Link from "next/link"
import RedirectButton from "../components/RedirectButton"
export default function Page() {
    return (
        <div>
        <h1>Bienvenue sur Next.js 15 !</h1>
        <p>Ceci est la page d'accueil.</p>
        <nav>
          <Link href="/about">À propos</Link> ||
          <Link href="/">Accueil</Link> ||
          <Link href="/dashboard">Tableau de bord</Link>
        </nav>
        <RedirectButton/>
      </div>
    )
}
```

Lorsqu'on accède à la page d'accueil **http://localhost:3000** et qu'on clique sur le bouton, on sera redirigé vers la page de dashboard.



7 Styles

En Next.js, nous avons plusieurs méthodes de styles.

7.1 Tailwind CSS

Par défaut, il utilise Tailwind CSS, qui est un framework de CSS.

Ici, nous utilisons les classes prédéfinies pour styliser nos composants.

8 CSS Modules

Les CSS Modules permettent de créer des classes CSS uniques. Ils ont une extension : **.module.css**.

Dans le fichier **app/components/Button.module.css** :

```
.button {
  padding: 10px 20px;
  background-color: blue;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

.button:hover {
  background-color: darkblue;
}
```

Dans le fichier **app/components/Button.tsx** :

```
import styles from './Button.module.css';

export default function Button({ children }: { children: React.ReactNode }) {
  return <button className={styles.button}>{children}</button>;
}
```

Nous allons utiliser cette classe dans le fichier **app/page.tsx** :

```
import Button from '../components/Button';

export default function Home() {
  return (
    <div>
      <h1>Bouton stylisé</h1>
      <Button>Clique-moi</Button>
    </div>
  );
}
```

9 Rendu et récupération des données

9.1 Récupération des données

Nous allons utiliser **fetch** pour la récupération des données depuis une **API** ou une **base de données**, en utilisant **async/await**. Dans la section suivante, nous l'utiliserons.

9.2 Client Side Render (CSR)

Client Side Render (CSR) signifie que le rendu est fait côté client (navigateur), c'est-à-dire que Next.js envoie une page HTML vide, puis la mise à jour de la page est faite dans le navigateur pour remplir la page.

Avant de continuer, nous allons parler d'un hook React appelé **useEffect()**. Il permet de gérer les effets de bord dans les composants fonctionnels.

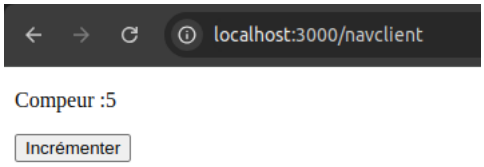
Nous avons plusieurs façons d'utiliser **useEffect()** :

- **Exécuter le code après chaque rendu** : Si on ne passe pas de tableau de dépendances, le code dans **useEffect** sera exécuté après chaque rendu du composant.

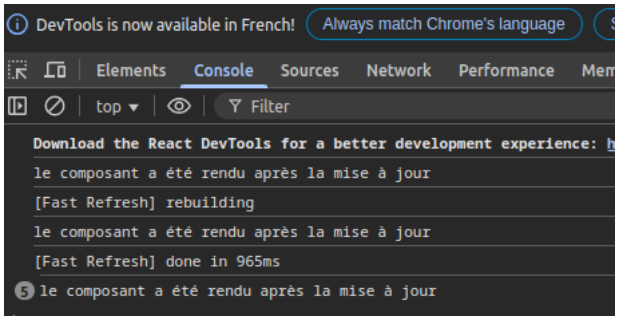
  Exemple : dans le fichier **/app/navclient/page.tsx** :

```
"use client"
import { useEffect, useState } from "react"

export default function EffectPage() {
    const [count, setCount] = useState(0)

    useEffect(() => {
        console.log("Le composant a été rendu après la mise à jour")
    })

    return (
        <div>
            <p>Compteur : {count} </p>
            <button onClick={() => setCount(count + 1)}>Incrémenter</button>
        </div>
    )
}
```

  Lorsqu'on accède à la route : **/navclient**



  On inspecte et on voit à chaque clic :



- **Exécuter le code après le premier rendu une seule fois** : Si on passe un tableau de dépendances vide, le code dans **useEffect** s'exécute une seule fois après le premier rendu du composant.

  Toujours dans le même fichier **/app/navclient/page.tsx** : on va modifier :

```
"use client"
import { useEffect, useState } from "react"

export default function EffectPage() {
    const [data, setData] = useState(null)

    useEffect(() => {
        console.log("Composant rendu")

        setTimeout(() => {
            setData("Données chargées")
        }, 5000);
    }, [])

    return (
        <div>
            {data ? data : "Chargement ..."}
        </div>
    )
}
```

  On accède toujours à **/navclient**, on verra que le message de la console s'affiche une seule fois, tandis que les données sont chargées après 5 secondes.

- **Exécuter le code lorsque les dépendances changent** : Si on passe un tableau avec des dépendances, le code dans **useEffect** s'exécute après le premier rendu et à chaque fois que les dépendances changent.

```
"use client"
import { useEffect, useState } from "react"

export default function EffectPage() {
    const [count, setCount] = useState(0)
    const [message, setMessage] = useState("")

    useEffect(() => {
        setMessage(`Le compteur est à ${count}`)
    }, [count])

    return (
        <div>
            <p>{message}</p>
            <button onClick={() => setCount(count + 1)}>Incrémenter</button>
        </div>
    )
}
```

Exemple de **CSR** : **app/csr/page.tsx**
Nous allons utiliser une API externe.

```
"use client"
import { useState, useEffect } from "react"
export default function CSRPage() {
    const [data, setData] = useState<string | null>(null)
    useEffect(() => {
        fetch("https://jsonplaceholder.typicode.com/posts/1")
          .then((res) => res.json())
          .then((post) => setData(post.title));
      }, []);

      return <h1>{data ? data : "Chargement..."}</h1>;
}
```

On accède à la route : **/csr**

9.3 Server Side Rendering (SSR)

Signifie que la page est générée sur le serveur à chaque requête puis envoyée au navigateur.

Exemple d'implémentation de SSR avec une API externe :
Créer le fichier : **app/ssr/page.tsx**

```
interface User {
    id: number,
    name: string
}

export default async function SSRPage(){
    const res = await fetch("https://jsonplaceholder.typicode.com/users",
     { cache: "no-store" });
    const users: User[] = await res.json()

    return (
        <div>
            <h1>Utilisateurs SSR</h1>
            <ul>
                {
                    users.map((user) => (
                        <li key={user.id}>{user.name}</li>
                    ))
                }
            </ul>
        </div>
    )
}
```

Accès à la route : **/ssr**

9.4 SSG (Static Site Generation)

Le **SSG (Static Site Generation)** signifie que la page est générée une seule fois au build, puis servie statique. À chaque requête, c'est la même page qui est envoyée.

Exemple de SSG, dans le fichier **app/ssg/page.tsx** :

```
interface User {
    id: number,
    name: string
}

export default async function SSGPage() {
    const res = await fetch('https://jsonplaceholder.typicode.com/users')
    const users: User[] = await res.json()

    return (
        <div>
            <h1>Utilisateurs SSG</h1>
            <ul>
                {
                    users.map((user) => (
                        <li key={user.id}>{user.name}</li>
                    ))
                }
            </ul>
        </div>
    )
}
```

Route : **/ssg**

9.5 ISR (Incremental Static Regeneration)

Permet de régénérer une page après un certain temps.

```
interface User {
    id: number,
    name: string
}

export default async function ISRPage() {
    const res = await fetch('https://jsonplaceholder.typicode.com/users',
        {
            next: { revalidate: 10 }
        }
    )

    const users: User[] = await res.json()

    return (
        <div>
            <h1>Utilisateurs ISR</h1>
            <ul>
                {
                    users.map((user) => (
                        <li key={user.id}>{user.name}</li>
                    ))
                }
            </ul>
```

```
        </div>
    )
}
```

Route : **/isr**

10 Mise en pratique avec un projet de gestion de tâches, avec ajout de certaines fonctionnalités

10.1 Introduction

Ce tutoriel vous guidera pas à pas dans la création d'une application de gestion de tâches avec Next.js. Nous allons commencer par la mise en place du projet, la création des composants de base, et l'ajout de fonctionnalités pour gérer les tâches.

10.2 Structure du projet

Après avoir créé le projet avec la commande suivante :

```
npx create-next-app@latest my-app
```

Voici la structure initiale du projet :

```
my-app/
- app/
  - layout.tsx
  - page.tsx
- components/  (à créer)
    - Header.tsx
    - Navbar.tsx
    - Sidebar.tsx
- public/
- styles/
- package.json
- next.config.js
```

10.3 Création des composants de base

Nous allons créer trois composants principaux :

- **Header.tsx** : Un en-tête pour afficher le titre de la page.
- **Navbar.tsx** : Une barre de navigation pour naviguer entre les pages.
- **Sidebar.tsx** : Une barre latérale pour afficher des liens et des actions.

10.3.1 Header.tsx

Ce composant affiche un titre et un espace pour un avatar ou une icône utilisateur.

```
// app/components/Header.tsx
export default function Header() {
  return (
    <header className="flex justify-between items-center mb-6">
      <h2 className="text-2xl font-semibold text-gray-800 dark:text-white">
```

```
        Tableau de Bord
      </h2>
      <div className="flex items-center gap-4">
        <div className="w-10 h-10 bg-gray-300 dark:bg-gray-600 rounded-full" />
      </div>
    </header>
  );
}
```

10.3.2 Navbar.tsx

Ce composant contient des liens pour naviguer entre les pages et des boutons pour la connexion et l'inscription.

```
// app/components/Navbar.tsx
import Link from "next/link";

export default function Navbar() {
  return (
    <nav className="bg-gray-600 text-white p-4 shadow-md">
      <div className="container mx-auto flex justify-between items-center">
        <h1 className="text-2xl font-semibold">TaskManager</h1>
        <div className="flex space-x-6">
          <Link href="/" className="hover:bg-blue-700 px-4 py-2 rounded-lg">
            Accueil
          </Link>
          <Link href="/about" className="hover:bg-blue-700 px-4 py-2 rounded
          lg">
            À propos
          </Link>
          <Link
            href="/dashboard"
            className="hover:bg-blue-700 px-4 py-2 rounded-lg"
          >
            Tableau de bord
          </Link>
          <Link
            href="/auth/login"
            className="bg-blue-500 hover:bg-blue-600 px-6 py-2 rounded-lg text-
            white transition"
          >
            Connexion
          </Link>
          <Link
            href="/auth/register"
            className="bg-green-500 hover:bg-green-600 px-6 py-2 rounded-lg
            text-white transition"
          >
            Inscription
          </Link>
          <button className="bg-red-500 hover:bg-red-600 px-4 py-2 rounded-lg
          text-white transition">
            Déconnexion
          </button>
```

```
        </div>
      </div>
    </nav>
  );
}
```

10.3.3 Sidebar.tsx

Ce composant fournit :

- Une barre latérale responsive
- Un menu de navigation avec icônes
- Un bouton de déconnexion fonctionnel
- Des liens de navigation optimisés

```
// app/components/Sidebar.tsx
"use client";

import { useState, useEffect } from "react";
import { Home, List, PlusCircle, LogOut } from "lucide-react";
import Link from "next/link";
import { useRouter } from "next/navigation";

export default function Sidebar() {
  const router = useRouter();

  const handleLogout = () => {
    localStorage.removeItem("user");
    router.push("/auth/login");
  };

  return (
    <aside className="w-64 bg-white dark:bg-gray-800 shadow-lg min-h-screen p-5
     flex flex-col justify-between">
      <div>
        <h1 className="text-2xl font-bold text-gray-800 dark:text-white mb-6">
          Gestion Tâches
        </h1>
        <nav className="space-y-4">
          <NavItem icon={Home} label="Accueil" href="/" />
          <NavItem icon={List} label="Tâches" href="/dashboard" />
          <NavItem
            icon={PlusCircle}
            label="Ajouter une tâche"
            href="/dashboard/add-task"
          />
        </nav>
      </div>

      <div className="mt-6 flex flex-col gap-4">
        <button
```

```
            onClick={handleLogout}
            className="flex items-center gap-2 text-red-500"
          >
            <LogOut size={20} /> Déconnexion
          </button>
        </div>
      </aside>
    );
}

function NavItem({
  icon: Icon,
  label,
  href,
}: {
  icon: any;
  label: string;
  href: string;
}) {
  return (
    <Link href={href} className="block">
      <div className="flex items-center gap-3 text-gray-700 dark:text-gray-300
      p-2 hover:bg-gray-200 dark:hover:bg-gray-700 rounded-lg cursor-pointer">
        <Icon size={20} />
        {label}
      </div>
    </Link>
  );
}
```

si vous avez remarqué la ligne

```
import { Home, List, PlusCircle, LogOut } from "lucide-react";
```

est souligné parce que lucide-react n'est pas installé .
lucide-react est une bibliothèque d'icônes pour React, qui fournit des icônes SVG légers et personnalisables. Elle est basée sur Lucide, une version communautaire améliorée de Feather Icons.

lancez la commande :

```
npm install lucide-react
```

si l'installation a marché l'erreur devrait disparaitre

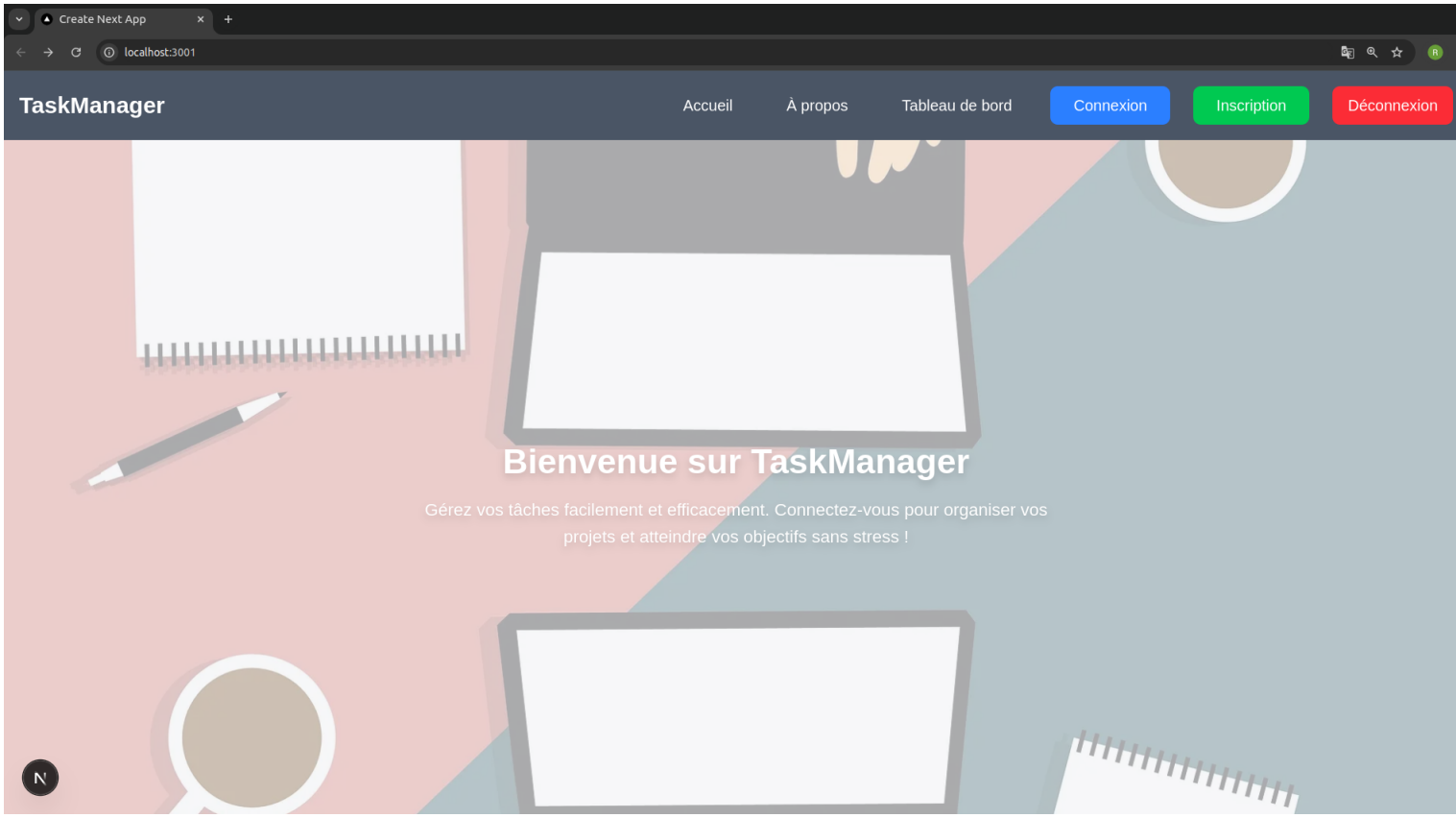
10.4 Modification de app/page.tsx

Nous allons ajouter le composant `NavBar` et une image dans la page d'accueil.

```
// app/page.tsx
import Link from "next/link";
import Image from "next/image";
import Navbar from "@/components/NavBar";

export default function HomePage() {
  return (
```

```
    <div className="bg-gray-100 dark:bg-gray-900 min-h-screen">
      {/* Navbar */}

      <Navbar/>
      {/* Page Content */}


    <div className="relative bg-gray-100 dark:bg-gray-900 min-h-screen">
      {/* Image de fond avec next/image */}
      <div className="absolute inset-0 opacity-30">
        <Image
          src="/images/tach.jpg"
          alt="Background Image"
          layout="fill"
          objectFit="cover"
          className="z-0"
        />
      </div>

      {/* Page Content */}
      <div className="relative z-10 flex flex-col items-center justify-center
      min-h-screen text-white p-6">
        <h1 className="text-4xl font-bold mb-4 text-center drop-shadow-lg">
          Bienvenue sur TaskManager
        </h1>
        <p className="text-lg mb-8 text-center max-w-2xl drop-shadow-lg">
          Gérez vos tâches facilement et efficacement. Connectez-vous pour
          organiser vos projets
          et atteindre vos objectifs sans stress !
        </p>
      </div>
    </div>




    </div>
  );
}
```

ce que vous devez avoir apres avoir appelé le composant navbar dans le page.tsx de app. Mais vous vous aurez pas de d'image pour le faire vous pouvez mettre un image dans le dossier /public/image(a creer) et venir le specifier dans page.tsx pour le voir



11 Ajout des routes et des pages

Nous allons créer les pages suivantes :

- **/about** : Page "À propos".
- **/dashboard** : Page du tableau de bord.
- **/auth/login** : Page de connexion.
- **/auth/register** : Page d'inscription.

11.1 Exemple de création de la page `/about`

```
// app/about/page.tsx
export default function About() {
  return (
    <div>
      <h1>À propos de TaskManager</h1>
      <p>Ceci est une application de gestion de tâches.</p>
    </div>
  );
}
```

11.2 Intégration des composants dans la page principale

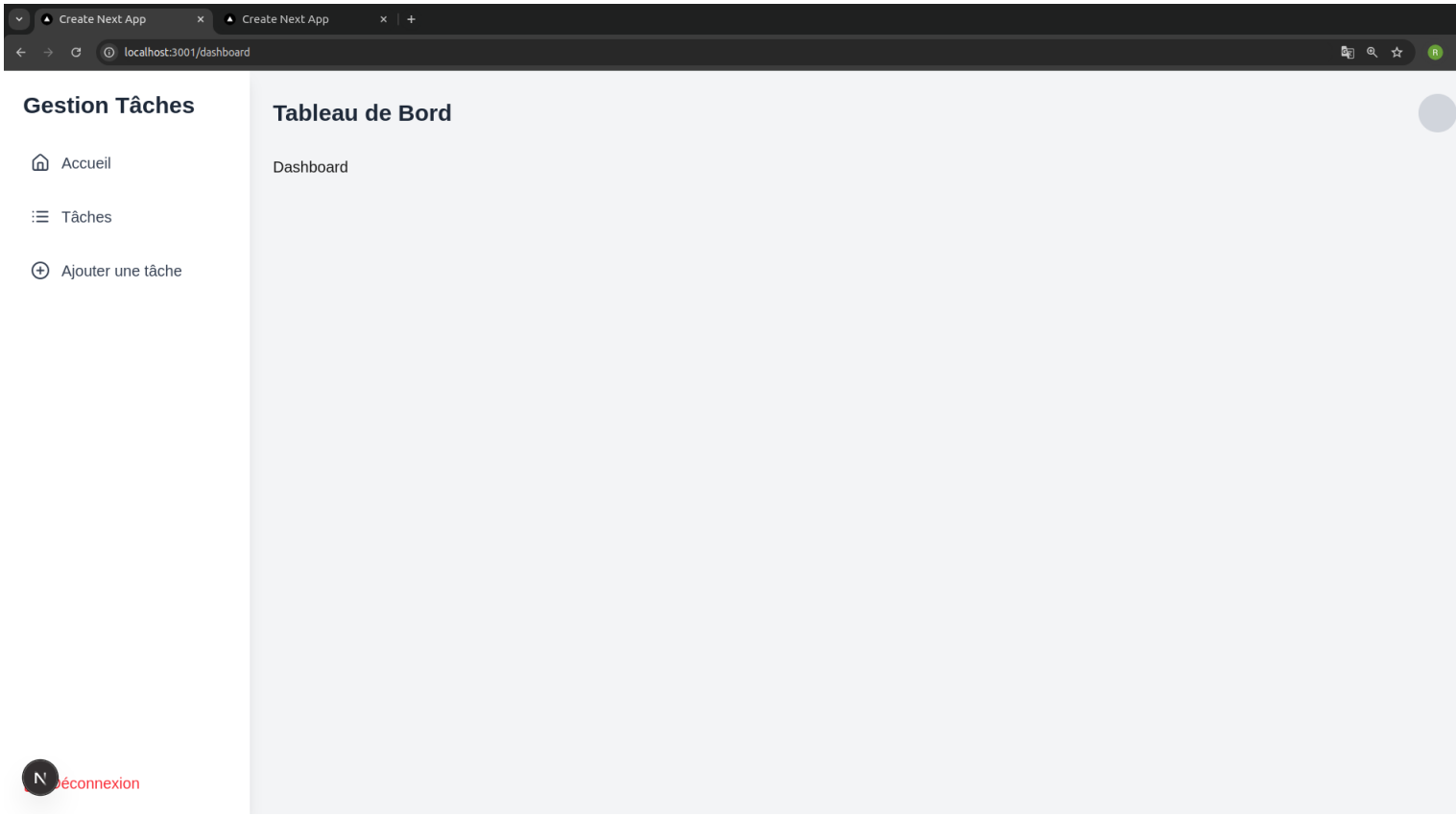
11.2.1 Modification de `app/dashboard/layout.tsx`

la page admin de notre projet sera la page de dashboard donc vous aller cree un layout.tsx pour partager la sidebar entre toutes les page de dashboard et les pages de ses sous repertoire .

Nous allons ajouter les composants `Sidebar` et `Header` dans le layout principal.

NB: la page.tsx de dashboard doit au moins avoir un code tsx a l'interieur sinon il aura une erreur

```tsx
// app/dashboard/layout.tsx
"use client"

import { Home, List, BarChart2, LogOut } from "lucide-react";
import Header from "@/components/Header";
import Sidebar from "@/components/Sidebar";
import Navbar from "@/components/NavBar";

import { useEffect, useState } from "react";
import { useRouter } from "next/navigation";

export default function DashboardLayout({ children }: { children: React.ReactNode }) {


  const [user, setUser] = useState(null);
  const router = useRouter();

   // useEffect(() => {
  //   const storedUser = localStorage.getItem("user");
  //   if (!storedUser) {
  //     router.push("/auth/login");
  //   } else {
  //     setUser(JSON.parse(storedUser));
  //   }
  // }, [router]);

  // if (!user) {
  //   return null;
  // }



  return (
    <div className=" flex min-h-screen bg-gray-100 dark:bg-gray-900">
      <Sidebar/>
      <main className="flex-1 p-6">
        <Header/>
        {children}
      </main>
    </div>
  );
}
```



12 Intégration de Prisma avec la base de données

12.1 Configuration initiale

12.1.1 Installation des dépendances

```
npm install prisma @prisma/client
npx prisma init
```

12.1.2 Schéma Prisma

Ajoutez ce schéma dans `prisma/schema.prisma` :

```
model User {
id Int @id @default(autoincrement())
name String
email String @unique
password String
tasks Task[]
}

model Task {
id Int @id @default(autoincrement())
title String
description String?
dueDate DateTime
completed Boolean @default(false)
user User @relation(fields: [userId], references: [id])
```

```
userId Int
}
```

12.2 Migrations et génération

12.2.1 Exécution des migrations

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

12.3 Utilisation dans Next.js

12.3.1 Initialisation du client Prisma

Créez `lib/prisma.ts` :

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

const prisma = new PrismaClient()
export default prisma
```

13 Pages d'authentification

13.1 Page d'inscription (`/app/auth/register/page.tsx`)

Ce code va implémenter un formulaire d'inscription d'utilisateur en React avec Next.js. Lorsqu'un utilisateur soumet le formulaire, les données sont envoyées à la fonction registerUser(), qui effectue l'inscription. En cas d'erreur, les messages d'erreur sont affichés ; sinon, l'utilisateur est redirigé vers la page de connexion.

```
"use client";

import React, { useState } from "react";
import { registerUser } from "../serviceAuth/authService";
import { useRouter } from "next/navigation";

export default function RegisterUser() {
  const router = useRouter();
  const [errors, setErrors] = useState<{name?:string, email?: string;

  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
      event.preventDefault();
      const formData = new FormData(event.currentTarget);
      const response = await registerUser(formData);

      if("error" in response) {
        setErrors(response.error);
      } else {
        router.push("/auth/login");
      }
  };
```

```
  return (
    <div className="flex justify-center items-center h-screen bg-gray-100">
      <form onSubmit={handleSubmit} className="bg-white p-6 rounded-lg shadow-
        <h2 className="text-2xl font-bold mb-4 text-gray-800">Inscription</h2>
        {errors.general && <p className="text-red-500">{errors.general}</p>}
        <div className="mb-3">
          <input name="name" type="text" placeholder="Nom"
                 className={`w-full p-2 border rounded mb-3 ${errors.password ?
          {errors.name && <p className="text-red-500 text-sm">{errors.name}</
        </div>
        <div className="mb-3">
          <input name="email" type="email" placeholder="Email"
                 className={`w-full p-2 border rounded mb-3 ${errors.email ?
          {errors.email && <p className="text-red-500 text-sm">{errors.email}</
        </div>
        <div className="mb-3">
          <input name="password" type="password" placeholder="Mot de passe"
                 className={`w-full p-2 border rounded mb-3 ${errors.password ?
          {errors.password && <p className="text-red-500 text-
        </div>

        <button type="submit" className="w-full bg-blue-500 text-white p-2
          S'inscrire
        </button>
      </form>
    </div>
  );
}
```

Explication général du code

13.1.1 Imports

Pour implémenter le formulaire d'inscription, nous avons besoin des imports suivants :

- `useState` : Pour gérer les erreurs d'inscription.
- `registerUser` : Pour envoyer les données au service d'authentification.
- `useRouter` : Pour rediriger l'utilisateur après inscription.

13.1.2 Gestion du formulaire (`handleSubmit`)

1. Empêche le rechargement de la page avec `event.preventDefault()`.
2. Récupère les données via `new FormData(event.currentTarget)`.
3. Envoie ces données à `registerUser()`.
4. Si une erreur est retournée, elle est stockée dans `setErrors()`.
5. Sinon, redirection vers `/auth/login`.

13.1.3 Affichage du formulaire

Le formulaire comprend :

- Trois champs : **Nom**, **Email**, **Mot de passe**.
- Un bouton **S'inscrire**.
- Affichage des erreurs sous chaque champ en cas de problème.

13.2 Service d'inscription (`/app/auth/serviceAuth/authService.ts`)

Ce code va définir une fonction serveur (registerUser) qui gère l'inscription d'un utilisateur dans une base de données avec Prisma. Il vérifie si l'utilisateur existe déjà, hache le mot de passe, et si tout est validé, crée un nouvel utilisateur dans la base de données.

pour haché le mot de passe on va installer

```
npm install bcryptjs
```

bcryptjs est une bibliothèque JavaScript qui permet de hacher des mots de passe de manière sécurisée. C'est une implémentation de l'algorithme bcrypt, spécialement conçu pour le stockage sécurisé des mots de passe.

```
"use server"

import prisma from "@/lib/prisma";
import bcrypt, { compare } from "bcryptjs";
import { revalidatePath } from "next/cache";
import { redirect } from "next/navigation";
//import { useRouter } from "next/navigation";

type AuthResponse =
        |{user : {id:number, name:string, email:string} }
        |{error : {name?:string, email?:string,password?:string,general?:string}};

export async function registerUser(formData : FormData)        Promise<AuthResponse>{

        const name = formData.get("name") as string
        const email = formData.get("email") as string
        const password = formData.get("password") as string

        if(!name || !email || !password) {
            return {
                error : {name : "le nom est requis", email:"l'email est
            }
        }

        const existingUser = await prisma.user.findUnique(
            {
                where : {email}
            }
        )

        if(existingUser) {
            return {
```

```
            error : {email:"l'email existe déjà"}
        }
    }

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


    const user = await prisma.user.create(
        {
            data : {name, email, password:hashedPassword}
        }
    )


    //redirect('/auth/login')


    return {
        user : {id:user.id, name:user.name,email:user.email}
    }

    }
```

Explication général du code

13.2.1 Imports

Les modules nécessaires pour l'inscription sont les suivants :

- `prisma` : Connexion à la base de données avec Prisma.
- `bcryptjs` : Pour hacher le mot de passe avant de l'enregistrer.
- `revalidatePath` et `redirect` de `next/navigation` : Pour effectuer une redirection après l'inscription (commenté ici).

13.2.2 Validation des Champs

La fonction vérifie les champs `name`, `email` et `password` du formulaire :

- Si l'un des champs est vide, une erreur est retournée.

13.2.3 Vérification de l'Existence de l'Utilisateur

La fonction vérifie si un utilisateur avec cet `email` existe déjà dans la base de données :

- Si l'email existe déjà, une erreur est retournée.

13.2.4 Hachage du Mot de Passe

Si l'utilisateur est valide, son mot de passe est haché avec `bcrypt.hash()` avant d'être enregistré.

13.2.5 Création de l'Utilisateur

Un nouvel utilisateur est créé dans la base de données à l'aide de Prisma.

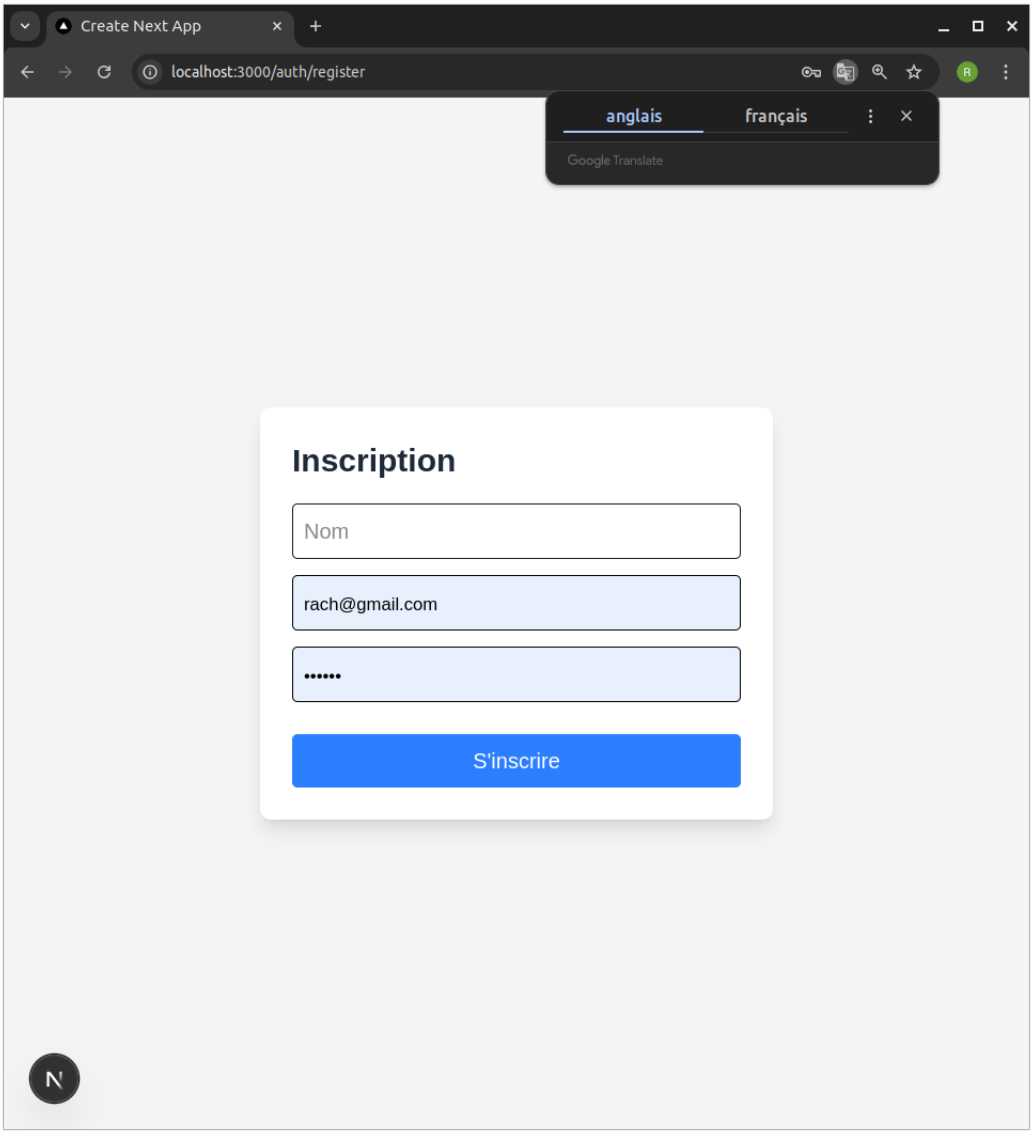
13.2.6 Retour du Résultat

En cas de succès :

- Un objet contenant les informations de l'utilisateur (`ID`, `name`, `email`) est retourné.

En cas d'erreur :

- Un objet d'erreur avec des messages spécifiques est retourné.



13.3 Page de connexion (`/app/auth/serviceAuth/authService.ts`)

Ce code va définir un composant LoginUser dans une application Next.js pour gérer la connexion des utilisateurs. Il envoie les informations d'identification via un formulaire à une fonction loginUser. Si la connexion réussit, il enregistre l'utilisateur dans le localStorage et redirige l'utilisateur vers la page d'accueil.

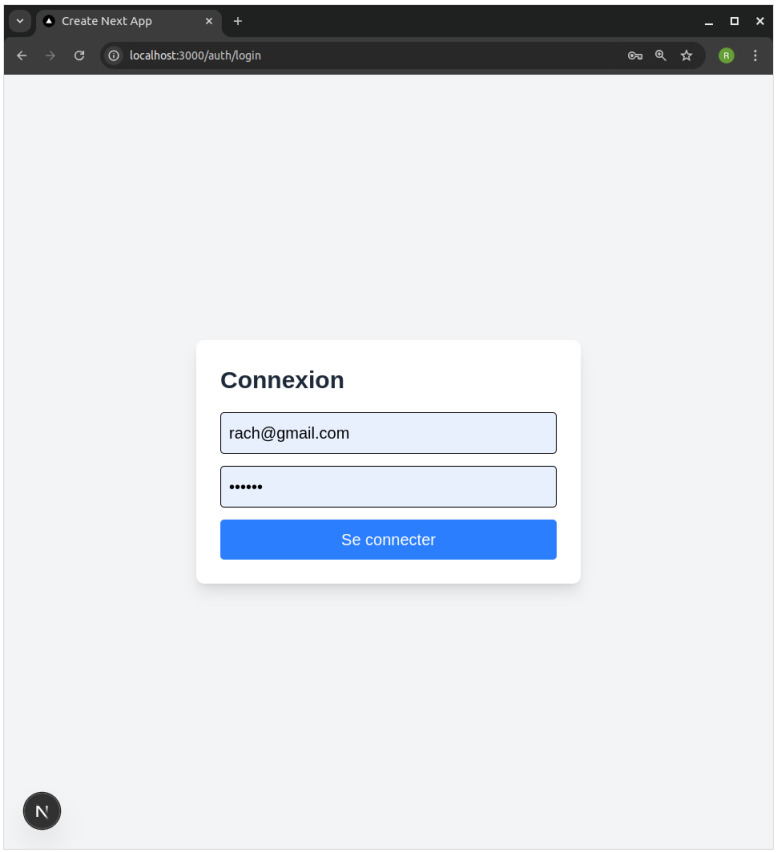
```
"use client";

import { useRouter } from "next/navigation";
import { useState } from "react";
import { loginUser } from "../serviceAuth/authService";
```

```
export default function LoginUser() {

    const [errors, setErrors] = useState<{ email?: string; password?: string;
    const router = useRouter();

    const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
        event.preventDefault();
        setErrors({});

        const formData = new FormData(event.currentTarget);
        const response = await loginUser(formData);

        if ("error" in response) {
            setErrors(response.error);
            return;
        }
        localStorage.setItem("user", JSON.stringify(response.user));
        router.push("/");
    };

    return (
        <div className="flex justify-center items-center h-screen bg-gray-100">
            <form onSubmit={handleSubmit} className="bg-white p-6 rounded-lg
                <h2 className="text-2xl font-bold mb-4 text-

                {errors.general && <p className="text-red-500 text-sm

                <div className="mb-3">
                    <input name="email" type="email" placeholder="Email"
                        className={`w-full p-2 border rounded ${errors.email ?
                         />
                    {errors.email && <p className="text-red-500 text-
                </div>

                <div className="mb-3">
                    <input name="password" type="password" placeholder="Mot de
                        className={`w-full p-2 border rounded ${errors.password
                         />
                    {errors.password && <p className="text-red-500 text-
                </div>

                <button type="submit" className="w-full bg-blue-500 text-white
            </form>
        </div>
    );
}
```

13.4 Service d'authentification (/app/auth/serviceAuth/authService.ts)

Ce code va définir une fonction serveur (loginUser) qui gère la connexion d'un utilisateur. Elle vérifie si les informations d'identification sont correctes (email et mot de passe) en les comparant avec les données dans la base de données. Si les informations sont valides, elle retourne les détails de l'utilisateur, sinon, elle retourne une erreur.

```
export async function loginUser(formData : FormData) : Promise<AuthResponse> {

    const email = formData.get('email') as string
    const password = formData.get('password') as string

    if(!email || !password) {
        return {
            error: {email:"email est requis", password:"le mot de passe est
        }
    }

    const user = await prisma.user.findUnique(
        {
            where : {email}
        }
    )

    if(!user || !(await compare(password, user.password))) {
        return {
            error : {general : "l'email ou le mot de passe est incorrect"}
        }
    }

    return {
        user : {id:user.id, name : user.name, email:user.email}
    }


}




}
```



après avoir implémenté tout sa, décommentez le code qui se trouve le layout de dashboard

Ce code est conçu pour protéger l'accès à une page du tableau de bord (dashboard) en vérifiant si l'utilisateur est connecté. Lorsqu'un utilisateur tente d'accéder à cette page, le code vérifie dans le localStorage si les informations de l'utilisateur sont présentes. Si l'utilisateur n'est pas connecté (pas d'informations dans le localStorage), il est redirigé vers la page de connexion.

```
useEffect(() => {
   const storedUser = localStorage.getItem("user");
   if (!storedUser) {
    router.push("/auth/login");
    } else {
      setUser(JSON.parse(storedUser));
  }
 }, [router]);

 if (!user) {
    return null;
  }
```

14 Système complet de gestion des tâches

14.1 Structure du service

Ce code va fournir un ensemble de fonctions utilisées pour gérer des tâches dans une application, incluant la création, la récupération, la mise à jour et la suppression des tâches. Il utilise Prisma pour interagir avec une base de données et Next.js pour la gestion de cache et la revalidation des chemins.

createTask permet de créer une tâche en enregistrant le titre, la description, la date d'échéance et l'utilisateur associé, puis actualise la page /dashboard.

getTasks récupère et renvoie toutes les tâches présentes dans la base de données sous forme de tableau.

getTaskById permet de récupérer une tâche spécifique en fonction de son ID.

updateTask modifie les informations d'une tâche existante, telles que le titre, la description, la date d'échéance, et son état de complétion, et actualise le tableau de bord.

deleteTask supprime une tâche par son ID et actualise également la page /dashboard pour refléter les modifications.

```ts
// /app/service/tacheService.ts
"use server";

import prisma from "@/lib/prisma";
import { revalidatePath } from "next/cache";

export async function createTask(data: {
  title: string;
  description: string;
  dueDate: string;
  userId: string
}) {
  try {
    await prisma.task.create({
      data: {
        title: data.title,
        description: data.description,
        dueDate: new Date(data.dueDate),
        completed: false,
        userId: Number(data.userId),
      },
    });
    revalidatePath("/dashboard");
    return { success: true };
  } catch (error) {
    return { success: false, message: "Erreur lors de l'ajout de la tâche." };
  }
}

export async function getTasks() {
  try {
    const tasks = await prisma.task.findMany();
    return { success: true, data: tasks };
  } catch (error) {
    return { success: false, data: [] };
  }
}

export async function getTaskById(taskId: number) {
  try {
```

```
    return await prisma.task.findUnique({ where: { id: taskId } });
  } catch (error) {
    return null;
  }
}

export async function updateTask(
  taskId: number,
  data: {
    title: string;
    description: string;
    dueDate: string;
    completed: boolean
  }
) {
  try {
    await prisma.task.update({
      where: { id: taskId },
      data: {
        title: data.title,
        description: data.description,
        dueDate: new Date(data.dueDate),
        completed: data.completed,
      },
    });
    revalidatePath("/dashboard");
    return { success: true };
  } catch (error) {
    return { success: false, message: "Erreur lors de la mise à jour." };
  }
}

export async function deleteTask(taskId: string) {
  try {
    await prisma.task.delete({ where: { id: Number(taskId) } });
    revalidatePath("/dashboard");
    return { success: true };
  } catch (error) {
    return { success: false, message: "Échec de la suppression." };
  }
}
```

14.2 Composant d'ajout de tâche

Ce code permet à un utilisateur authentifié d'ajouter une tâche via un formulaire dans une application Next.js. Lors de la soumission, les données sont envoyées à la fonction createTask() pour être enregistrées. Si la tâche est créée avec succès, l'utilisateur est redirigé vers le tableau de bord. Si une erreur survient, un message d'erreur est affiché.

pour la gestion du formulaire et la validation des champs du formulaire on va utiliser **Formik**(Formik est une bibliothèque populaire pour gérer les formulaires dans des applications React.) et **Yup**(Yup est une bibliothèque de validation de schémas, souvent utilisée avec Formik pour valider les

données de formulaires.)

installation

```
npm install formik yup
```

```
// /app/dashboard/add-task/page.tsx


"use client";
import { useEffect, useState } from "react";
import { useRouter } from "next/navigation";
import { Formik, Form, Field, ErrorMessage } from "formik";
import * as Yup from "yup";
// { createTask } from "./createTask";
import { createTask } from "@/app/service/tacheService";
export default function AddTask() {
  const router = useRouter();
  const [userId, setUserId] = useState<string | null>(null);
  const [minDate, setMinDate] = useState<string>("");

  useEffect(() => {
    const user = localStorage.getItem("user");
    if (user) {
      setUserId(JSON.parse(user).id);
    }

    const today = new Date().toISOString().split("T")[0];
    setMinDate(today);
  }, []);

  const validationSchema = Yup.object({
    title: Yup.string().required("Le titre est requis."),
    description: Yup.string().required("La description est requise."),
    dueDate: Yup.date()
      .min(new Date().toISOString().split("T")[0], "La date doit être à partir d'aujourd'hui
      .required("La date d'échéance est requise."),
  });

  const  handlerSubmit = async  (values, { setSubmitting, setErrors }) => {
    if (!userId) {
      setErrors({ title: "Utilisateur non authentifié." });
      setSubmitting(false);
      return;
    }

    const response = await createTask({ ...values, userId });

    if (response.success) {
      router.push("/dashboard");
    } else {
      setErrors({ title: response.message || "Erreur inconnue." });
    }
```

```
    setSubmitting(false);
  }

  return (
    <div className="max-w-2xl mx-auto bg-white dark:bg-gray-800 p-8 rounded-lg
      <h2 className="text-2xl font-semibold text-gray-800 dark:text-white

      <Formik
        initialValues={{ title: "", description: "", dueDate: "" }}
        validationSchema={validationSchema}
        onSubmit={handlerSubmit}
      >
        {({ isSubmitting }) => (
          <Form className="space-y-4">
            <div>
              <label className="block text-gray-700 dark:text-gray-300 font-
              <Field
                type="text"
                name="title"
                className="w-full mt-1 p-2 border rounded-lg dark:bg-gray-700
              />
              <ErrorMessage name="title" component="p" className="text-red-500
            </div>

            <div>
              <label className="block text-gray-700 dark:text-gray-300 font-
              <Field
                as="textarea"
                name="description"
                className="w-full mt-1 p-2 border rounded-lg dark:bg-gray-700
              />
              <ErrorMessage name="description" component="p" className="text-
            </div>

            <div>
              <label className="block text-gray-700 dark:text-gray-300 font-
              <Field
                type="date"
                name="dueDate"

                className="w-full mt-1 p-2 border rounded-lg dark:bg-gray-700
              />
              <ErrorMessage name="dueDate" component="p" className="text-
            </div>

            <button
              type="submit"
              disabled={isSubmitting}
              className="w-full bg-blue-500 hover:bg-blue-600 text-white font-
            >
              {isSubmitting ? "Ajout en cours..." : "Ajouter la Tâche"}
            </button>
          </Form>
```

```
        )}
      </Formik>
    </div>
  );
}
```

Explication général du code

14.2.1 État local

Les états locaux utilisés dans la gestion de la tâche sont les suivants :

- `userId` : Cet état est utilisé pour stocker l'ID de l'utilisateur actuel, récupéré depuis `localStorage`. Si l'utilisateur est authentifié, son ID est disponible dans cet état.
- `minDate` : Cet état est utilisé pour stocker la date minimale autorisée pour la tâche. Il est initialisé avec la date actuelle, empêchant ainsi de sélectionner une date passée pour la tâche.

14.2.2 Effet `useEffect`

Lors du montage du composant, `useEffect` vérifie si l'utilisateur est authentifié en lisant l'ID de l'utilisateur dans `localStorage` :

- Si un utilisateur est trouvé, son ID est stocké dans `userId`.
- La date minimale pour la date d'échéance (`minDate`) est définie à la date du jour, ce qui garantit que la date d'échéance ne peut pas être dans le passé.

14.2.3 Validation des données avec `Yup`

Un schéma de validation est défini avec `Yup` pour vérifier les conditions suivantes :

- Le titre de la tâche est requis.
- La description de la tâche est requise.
- La date d'échéance doit être une date future (à partir d'aujourd'hui).

14.2.4 Soumission du formulaire

Lors de la soumission du formulaire, la fonction de soumission vérifie les étapes suivantes :

- Si l'utilisateur n'est pas authentifié, une erreur est affichée et la soumission est arrêtée.
- Si l'utilisateur est authentifié, les données de la tâche (titre, description, date d'échéance) sont envoyées au backend pour être enregistrées.
- Si la tâche est créée avec succès, l'utilisateur est redirigé vers le tableau de bord (`/dashboard`).
- En cas d'erreur, un message d'erreur est affiché à l'utilisateur.

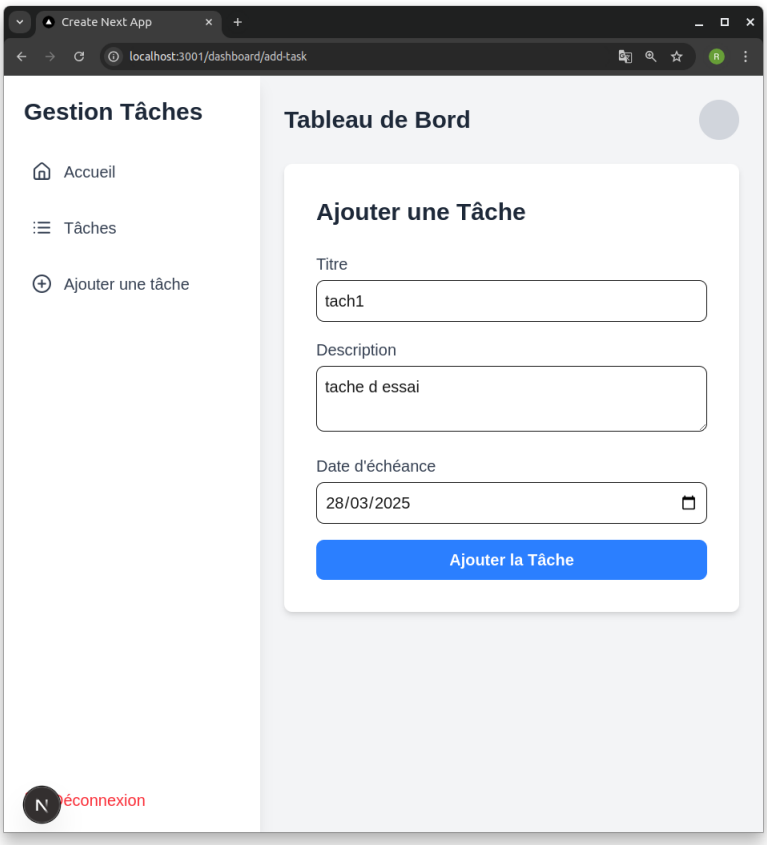
14.2.5 Affichage du formulaire

Le formulaire est géré avec `Formik`, ce qui permet de suivre l'état du formulaire, de gérer la validation et la soumission de manière fluide. Il comprend les champs suivants :

- **Titre** : Un champ texte pour entrer le titre de la tâche.
- **Description** : Un champ de texte pour entrer la description de la tâche.
- **Date d'échéance** : Un champ de type date où l'utilisateur peut sélectionner la date d'échéance de la tâche. La date minimale est fixée à la date actuelle.

Des messages d'erreur sont affichés sous chaque champ en cas de validation échouée. Un bouton **Ajouter la Tâche** permet de soumettre le formulaire, et ce bouton est désactivé pendant la soumission.

accédé au tableau de bord et cliquez sur ajouter tache de sidebar



verifier tout les controlles et validez si tout marche bien vous serez redirige vers le /dashboard qui devait normalement contenir le tableau des taches donc on va y remedier dans le prochain point

14.3 Composant de liste des tâches

Le composant **TaskList** est une interface React permettant aux utilisateurs de gérer leurs tâches de manière dynamique. Il récupère et affiche la liste des tâches depuis un backend, avec des détails tels que le titre, la description, la date d'échéance et le statut (complet/incomplet). Les utilisateurs peuvent rechercher des tâches par titre, les modifier ou les supprimer avec une confirmation via une boîte de dialogue modale (implémentée avec **Radix UI** pour une accessibilité optimale).

Rôle de Radix UI :

Radix UI fournit des composants accessibles et personnalisables, comme la boîte de dialogue de confirmation utilisée pour la suppression des tâches. Contrairement à des solutions comme react-modal, Radix UI se concentre sur la sémantique et le contrôle fin du comportement (gestion du focus, fermeture au clic externe, etc.), tout en restant léger et compatible avec Tailwind CSS

Installation de Radix UI :

```
npm install radix-ui @radix-ui/react-dialog  # Pour les boîtes de dialogue
```

```tsx
// /components/TaskList.tsx
"use client";

import { useState, useEffect } from "react";
import { getTasks, deleteTask } from "@/app/service/tacheService";

import { CheckCircle, Edit, Trash,  } from "lucide-react";
import { useRouter } from "next/navigation";
import * as Dialog from "@radix-ui/react-dialog";




interface Task {
  id: string;
  title: string;
  description: string | null;
  dueDate: Date;
  completed: boolean;
  userId: string;

}


export default function TaskList() {
  const [tasks, setTasks] = useState<Task[]>([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState("");
  const [searchTerm, setSearchTerm] = useState("");
  const [filteredTasks, setFilteredTasks] = useState<Task[]>([]);
  const [searchQuery, setSearchQuery] = useState("");
  const [selectedTask, setSelectedTask] = useState<Task | null>(null);
  const [isDialogOpen, setIsDialogOpen] = useState(false);




  const router = useRouter()
  useEffect(() => {
    const fetchTasks = async () => {
      setLoading(true);
      setError("");
```

```
      const user = localStorage.getItem("user");
      if (!user) {
        setError("Utilisateur non authentifié.");
        setLoading(false);
        return;
      }


      const response = await getTasks();

      if (response.success) {
        setTasks(response.data);
      } else {
        setError("Erreur lors du chargement des tâches.");
      }
      setLoading(false);
    };

    fetchTasks();
  }, []);

  const handleDeleteClick = (task: Task) => {
    setSelectedTask(task);
    setIsDialogOpen(true);
  };

 // Gestion de la recherche
 useEffect(() => {
  const filtered = tasks.filter((task) =>
    task.title.toLowerCase().includes(searchQuery.toLowerCase())
  );
  setFilteredTasks(filtered);
}, [searchQuery, tasks]);




const confirmDelete = async () => {
  if (!selectedTask) return;
  const response = await deleteTask(selectedTask.id);
  if (response.success) {
    setTasks((prev) => prev.filter((task) => task.id !== selectedTask.id));
  } else {
    setError(response.message || "Échec de la suppression.");
  }
  setIsDialogOpen(false);
};




  return (
    <div className="bg-white dark:bg-gray-800 p-5 rounded-lg shadow-md">
```

```
    <h2 className="text-2xl font-semibold text-gray-800 dark:text-white

    <div className="mb-5">
    <input
     type="text"
     placeholder="Rechercher une tâche..."
     value={searchQuery}
     onChange={(e) => setSearchQuery(e.target.value)}
     className="mb-4 p-2 w-full border border-gray-300 rounded-lg"
    />
    </div>

    {error && <p className="text-red-500">{error}</p>}
    {loading && <p className="text-gray-500">Chargement...</p>}

    <table className="min-w-full bg-white dark:bg-gray-700 rounded-lg shadow-
      <thead>
        <tr>
          <th className="py-3 px-6 text-left text-sm font-semibold text-
          <th className="py-3 px-6 text-left text-sm font-semibold text-
          <th className="py-3 px-6 text-left text-sm font-semibold text-
          <th className="py-3 px-6 text-left text-sm font-semibold text-
          <th className="py-3 px-6 text-left text-sm font-semibold text-
        </tr>
      </thead>
      <tbody>
        {filteredTasks.map((task) => (
          <tr
            key={task.id}
            className={`border-t ${task.completed ? "bg-green-100 dark:bg-
          >
            <td className="py-4 px-6 text-sm text-gray-700 dark:text-
            <td className="py-4 px-6 text-sm text-gray-600 dark:text-
            <td className="py-4 px-6 text-sm text-gray-600 dark:text-
            <td className="py-4 px-6 text-sm">
              <div className={`flex items-center gap-2`}>
                <CheckCircle
                  size={20}
                  className={`p-2 rounded-full ${task.completed ? "bg-
                />
                <span className={`text-sm ${task.completed ? "text-green-500"          : "t
                  {task.completed ? "Terminée" : "En cours"}
                </span>
              </div>
            </td>
            <td className="py-4 px-6 text-sm flex gap-2">
              <button onClick={() => router.push(`/dashboard/edit-task/$
                <Edit size={20} />
              </button>
              <button onClick={() => handleDeleteClick(task) } className="p-2          bg-r
                <Trash size={20} />
              </button>
            </td>
```

```
            </tr>
          ))}
        </tbody>
      </table>



      {/* Boîte de dialogue Radix UI */}
      <Dialog.Root open={isDialogOpen} onOpenChange={setIsDialogOpen}>
        <Dialog.Portal>
          <Dialog.Overlay className="fixed inset-0 bg-black/30 backdrop-blur-
          <Dialog.Content className="fixed top-1/2 left-1/2 transform -
            <Dialog.Title className="text-lg font-bold text-gray-800 dark:text-
              Confirmation de suppression
            </Dialog.Title>
            <Dialog.Description className="text-gray-600 dark:text-gray-300
              Êtes-vous sûr de vouloir supprimer cette tâche ?
            </Dialog.Description>
            <div className="mt-4 flex justify-end gap-3">
              <Dialog.Close asChild>
                <button className="px-4 py-2 bg-gray-300 dark:bg-gray-700 text-
                  Annuler
                </button>
              </Dialog.Close>
              <button
                onClick={confirmDelete}
                className="px-4 py-2 bg-red-600 text-white rounded"
              >
                Supprimer
              </button>
            </div>
          </Dialog.Content>
        </Dialog.Portal>
      </Dialog.Root>
    </div>
  );
}
```

Explication général du code

14.3.1 État local

Les états locaux utilisés dans la gestion des tâches sont les suivants :

- `tasks` : Un tableau qui contient toutes les tâches récupérées depuis le backend.
- `loading` : Un booléen qui indique si les données sont en train de se charger.
- `error` : Une chaîne de caractères qui contient un message d'erreur si quelque chose ne va pas pendant le chargement des tâches.
- `searchQuery` : Une chaîne de caractères pour stocker la recherche de l'utilisateur.
- `filteredTasks` : Un tableau qui contient les tâches filtrées en fonction de la recherche.

- `selectedTask` : L'objet de la tâche actuellement sélectionnée pour la suppression.
- `isDialogOpen` : Un booléen pour gérer l'ouverture et la fermeture de la boîte de dialogue de confirmation de suppression.

14.3.2 Récupération des tâches avec `useEffect`

Lors du premier rendu, le `useEffect` se déclenche pour récupérer les tâches via le service `getTasks`. Les étapes suivantes sont réalisées :

- Si l'utilisateur n'est pas authentifié (pas de données dans le `localStorage`), un message d'erreur est affiché.
- Si les tâches sont récupérées avec succès, elles sont stockées dans l'état `tasks`.
- En cas d'erreur, un message d'erreur est affiché.

14.3.3 Gestion de la recherche

Un autre `useEffect` est utilisé pour filtrer les tâches à chaque changement dans le champ de recherche (`searchQuery`) :

- Les tâches sont filtrées en fonction du titre de la tâche, et seules celles qui correspondent à la recherche sont affichées dans `filteredTasks`.

14.3.4 Suppression d'une tâche

Lorsqu'un utilisateur clique sur le bouton de suppression, la fonction `handleDeleteClick` est appelée :

- Elle ouvre la boîte de dialogue de confirmation de suppression et stocke la tâche sélectionnée dans `selectedTask`.
- Lorsque l'utilisateur confirme la suppression dans la boîte de dialogue, la fonction `confirmDelete` envoie la demande de suppression via le service `deleteTask`.
- Si la suppression est réussie, la tâche est retirée de l'état `tasks`. Sinon, un message d'erreur est affiché.

14.3.5 Affichage des tâches

Les tâches filtrées (`filteredTasks`) sont affichées dans un tableau. Chaque ligne contient les éléments suivants :

- **Titre** : Le titre de la tâche.
- **Description** : La description (si disponible).
- **Date d'échéance** : La date d'échéance, formatée au format de la locale de l'utilisateur.
- **Statut** : Le statut de la tâche, avec une indication visuelle de si elle est terminée ou en cours.
- **Actions** : Un bouton d'édition et un bouton de suppression.

Les tâches terminées sont affichées avec un fond vert et les tâches en cours avec un fond gris.

14.3.6 Boîte de dialogue Radix UI

Lorsque l'utilisateur clique sur le bouton de suppression, une boîte de dialogue apparaît pour confirmer l'action. Le composant `Dialog` de Radix UI est utilisé pour créer cette boîte de dialogue :

- La boîte de dialogue comprend une superposition sombre et un contenu centré.
- L'utilisateur peut soit annuler l'opération, soit confirmer la suppression de la tâche.

14.3.7 Comportement de la boîte de dialogue

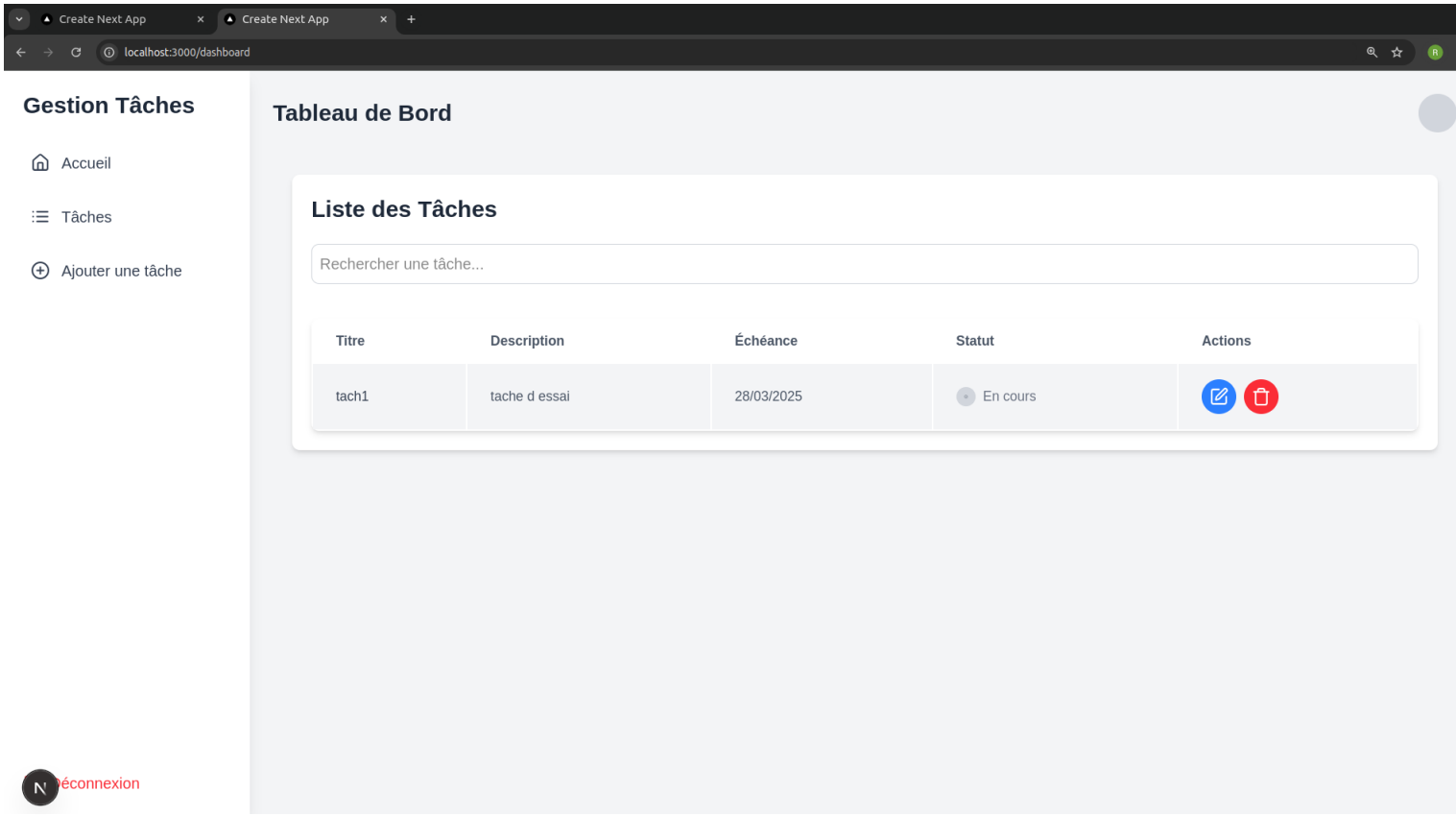
La boîte de dialogue est contrôlée par l'état `isDialogOpen`, qui est mis à jour pour ouvrir ou fermer la boîte de dialogue en fonction de l'action de l'utilisateur.

14.4 Page principale du tableau de bord

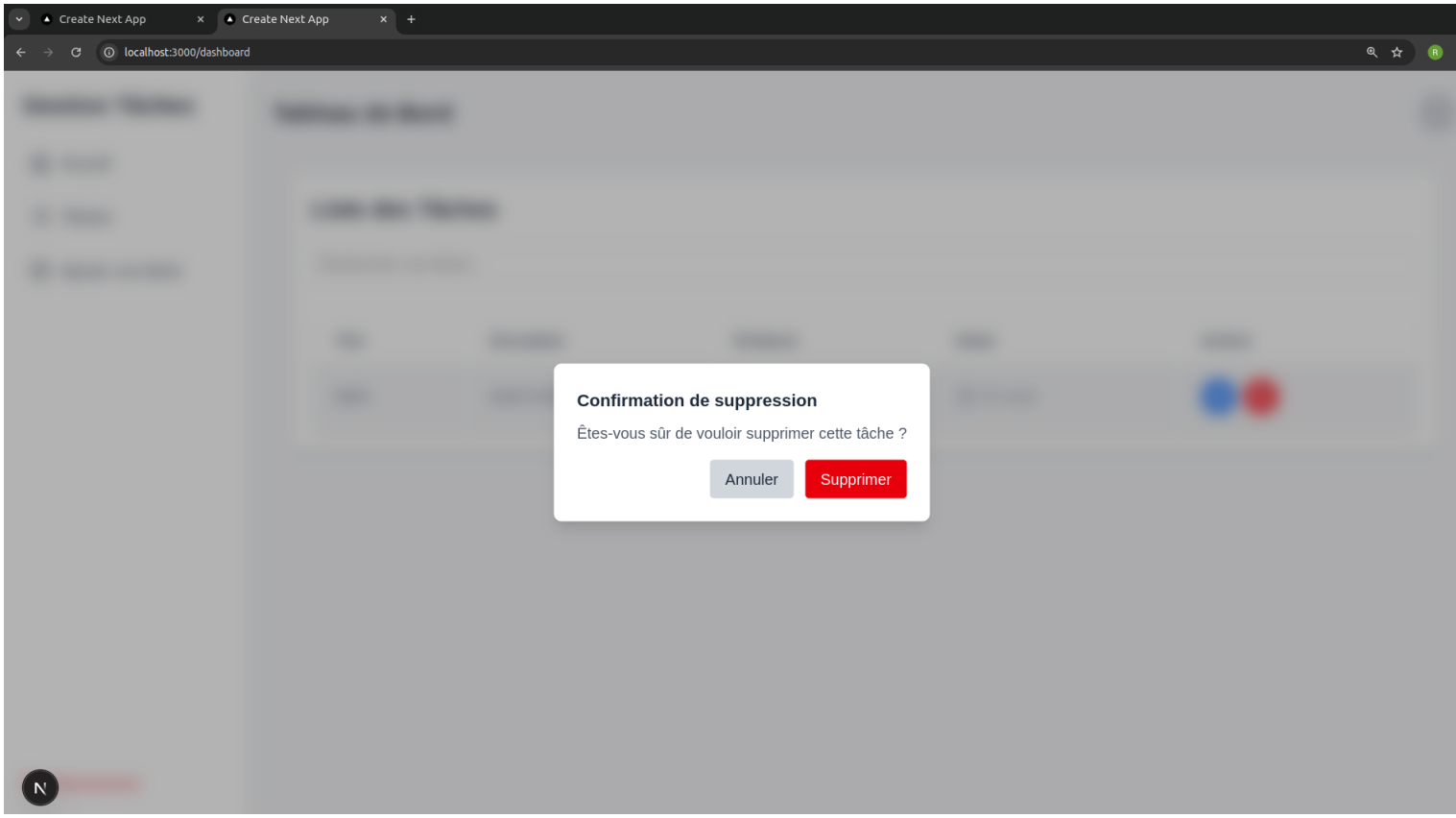
Le composant DashboardPage est destiné à être la page principale du tableau de bord de l'application. Il sert de point d'entrée pour l'affichage de la liste des tâches de l'utilisateur. Cette page

```tsx
// /app/dashboard/page.tsx
import TaskList from "@/components/TaskList";

export default function DashboardPage() {
  return (
    <div className="p-5">
      <TaskList />
    </div>
  );
}
```

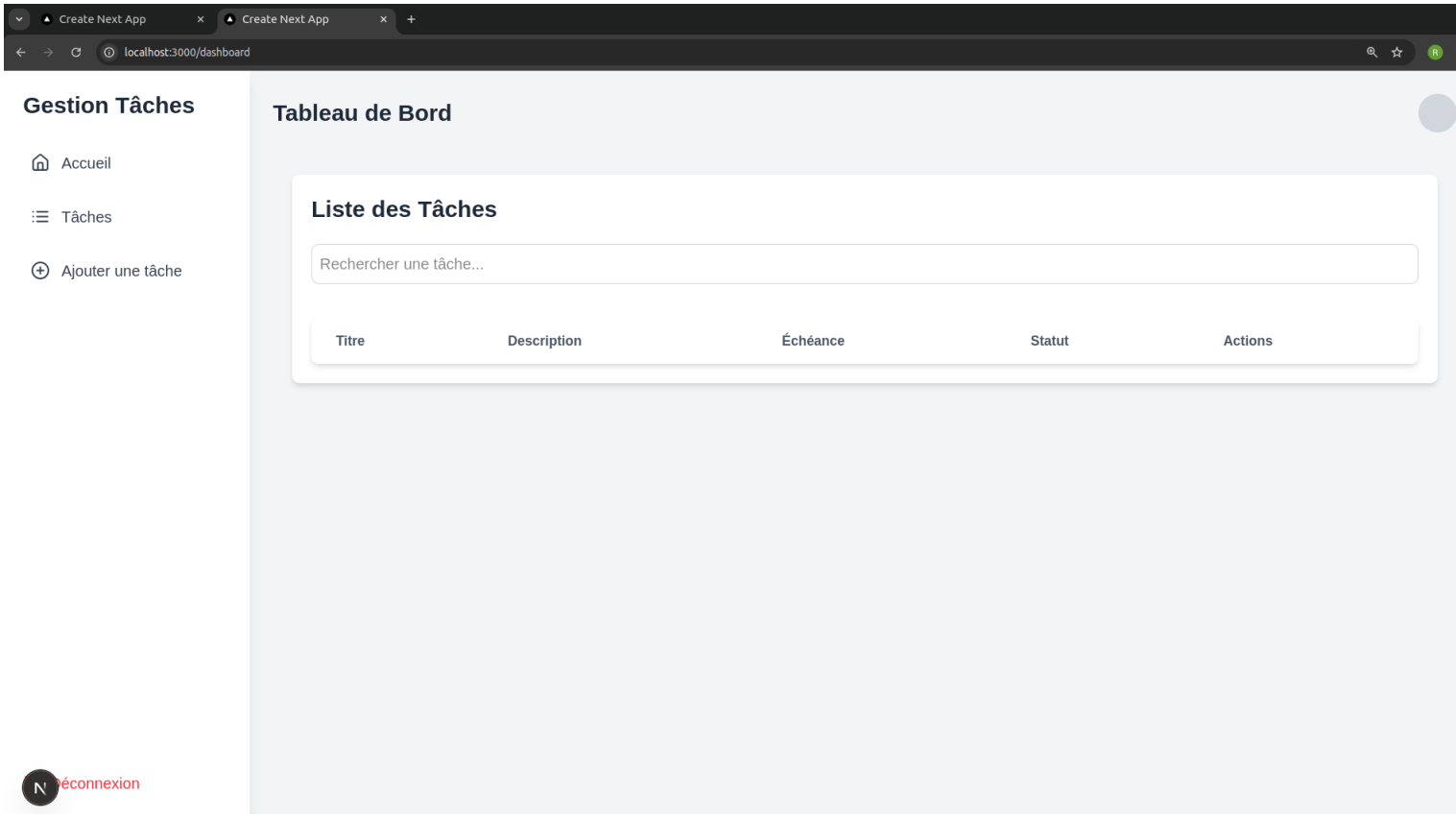
apres avoir ajouté le composant TaskList dans /app/dashboard/page.tsx vous verrez la tache ajoutée



nous allons profité pour tester le button de suppression
vous cliquez sur l'icon supprimer



vous confirmez et vous verrez sa



vous pouvez aussi testez le filtreur

14.5 Page d'édition de tâche

Le composant `EditTaskPage` est utilisé pour afficher un formulaire permettant à l'utilisateur de modifier une tâche existante. Lorsqu'un utilisateur sélectionne une tâche à modifier, ce composant permet de :

Fonctionnalités principales

- **Récupérer les données de la tâche** : Le composant récupère les informations de la tâche via une requête côté serveur en utilisant l'ID de la tâche. Cette récupération permet de pré-remplir le formulaire avec les valeurs actuelles de la tâche.
- **Afficher un formulaire pré-rempli** : Le formulaire est initialisé avec les données actuelles de la tâche, telles que le titre, la description, la date d'échéance et l'état de la tâche. L'utilisateur peut visualiser et modifier ces informations.
- **Modifier les informations de la tâche** : L'utilisateur peut modifier les détails de la tâche, tels que le titre, la description, la date d'échéance et l'état de complétion de la tâche.
- **Valider le formulaire** : Le formulaire est validé côté client à l'aide de la bibliothèque `Yup`. Cela garantit que toutes les informations saisies sont complètes et valides avant la soumission.
- **Soumettre les modifications** : Après modification, l'utilisateur peut soumettre le formulaire pour mettre à jour la tâche. Si la mise à jour est réussie, l'utilisateur est redirigé vers la page du tableau de bord.

Interface utilisateur

Le formulaire de modification est conçu de manière simple et intuitive, permettant à l'utilisateur de naviguer facilement entre les champs à remplir. Les champs incluent :

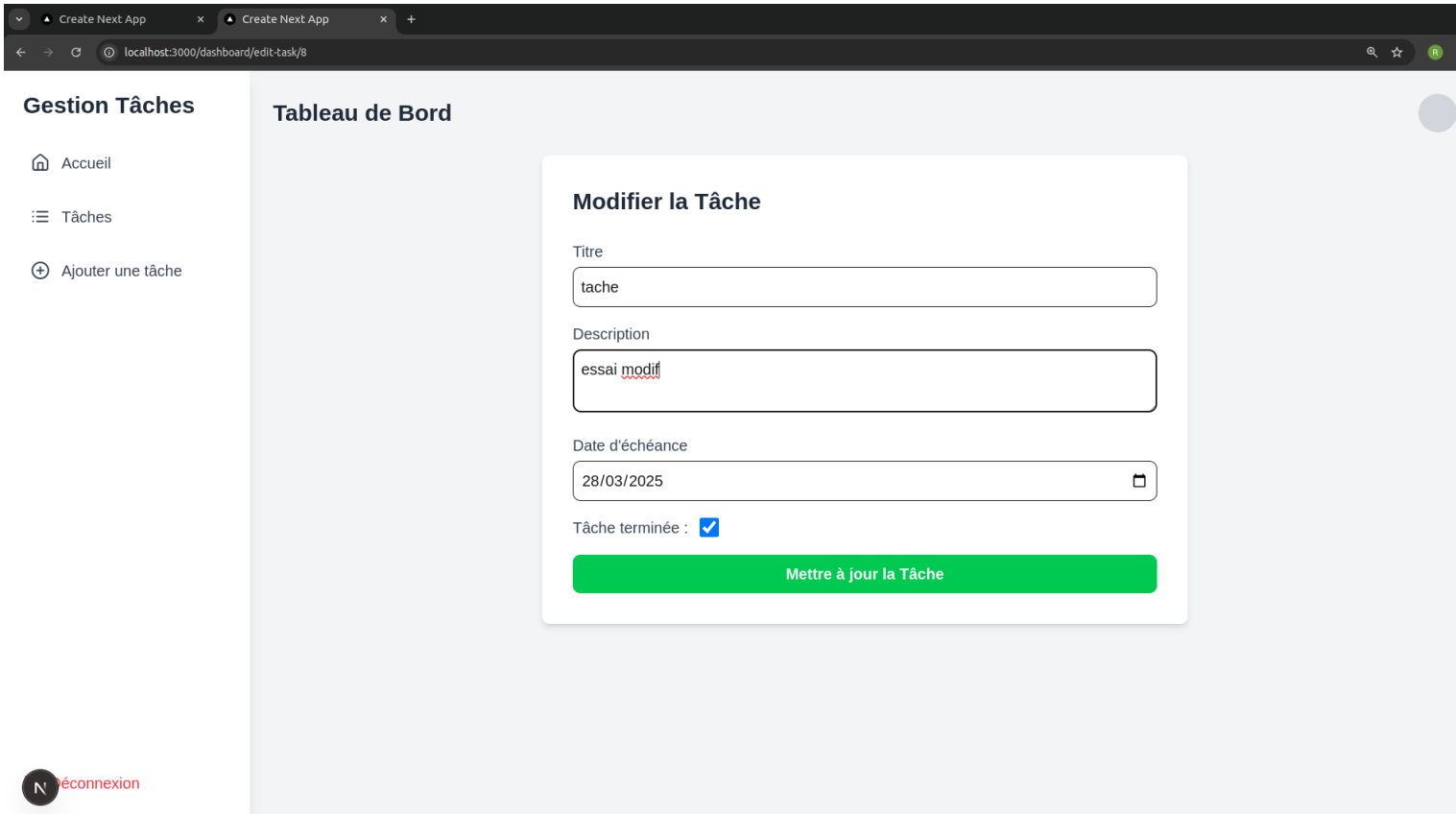
- **Titre** : Champ de texte pour saisir le titre de la tâche.
- **Description** : Champ de texte pour saisir une description détaillée de la tâche.
- **Date d'échéance** : Sélecteur de date pour choisir une nouvelle date d'échéance.
- **État de la tâche** : Sélecteur pour définir l'état actuel de la tâche (en cours ou terminée).

```tsx
// /app/dashboard/edit-task/[id]/page.tsx
"use client";

import {use, useEffect, useState } from "react";
import { useRouter } from "next/navigation";
import { Formik, Form, Field, ErrorMessage } from "formik";
import * as Yup from "yup";
import {getTaskById, updateTask } from "@/app/service/tacheService";

export default function EditTaskPage({ params }: { params: Promise<{ id: string }>  }) {
  const router = useRouter();    const today = new Date().toISOString().split("T")[0];

  const {id} = use(params);
  const taskId = Number(id)
  const [task, setTask] = useState<any>(null);
  const [loading, setLoading] = useState(true);
  const [minDate, setMinDate] = useState<string>("");

  useEffect(() => {
```

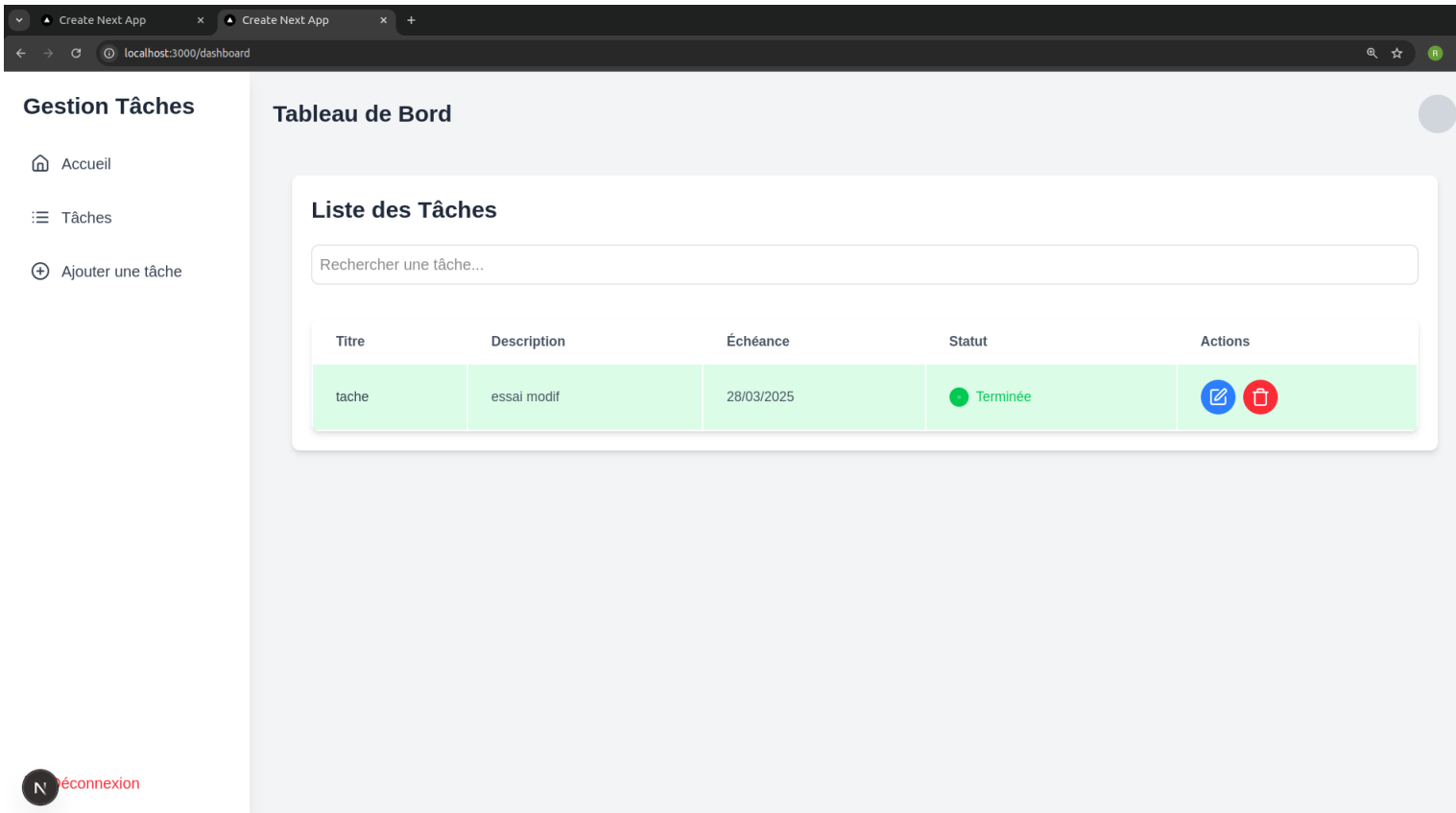
```
  async function fetchTask() {
    const data = await getTaskById(taskId);
    if (!data) {
      router.push("/not-found");
    } else {
      setTask(data);
    }
    setLoading(false);
  }

  fetchTask();
  setMinDate(new Date().toISOString().split("T")[0]);
}, [taskId, router]);

//  Validation du formulaire
const validationSchema = Yup.object({
  title: Yup.string().required("Le titre est requis."),
  description: Yup.string().required("La description est requise."),
  dueDate: Yup.date()
    .min(new Date().toISOString().split("T")[0], "La date doit être à partir
    .required("La date d'échéance est requise."),
});

if (loading) {
  return <p>Chargement...</p>;
}

return (
  <div className="max-w-2xl mx-auto bg-white dark:bg-gray-800 p-8 rounded-lg
    <h2 className="text-2xl font-semibold text-gray-800 dark:text-white

    <Formik
      initialValues={{
        title: task.title || "",
        description: task.description || "",
        dueDate: task.dueDate ? new
        completed: task.completed || false,
      }}
      validationSchema={validationSchema}
      onSubmit={async (values, { setSubmitting, setErrors }) => {
        const response = await updateTask(taskId, values);

        if (response.success) {
          router.push("/dashboard");
        } else {
          setErrors({ title: response.message || "Erreur lors de la mise à
        }

        setSubmitting(false);
      }}
    >
      {({ isSubmitting }) => (
        <Form className="space-y-4">
```

```
            <div>
              <label className="block text-gray-700 dark:text-gray-300 font-
              <Field
                type="text"
                name="title"
                className="w-full mt-1 p-2 border rounded-lg dark:bg-gray-700
              />
              <ErrorMessage name="title" component="p" className="text-red-500
            </div>

            <div>
              <label className="block text-gray-700 dark:text-gray-300 font-
              <Field
                as="textarea"
                name="description"
                className="w-full mt-1 p-2 border rounded-lg dark:bg-gray-700
              />
              <ErrorMessage name="description" component="p" className="text-
            </div>

            <div>
              <label className="block text-gray-700 dark:text-gray-300 font-
              <Field
                type="date"
                name="dueDate"
                min={minDate}
                className="w-full mt-1 p-2 border rounded-lg dark:bg-gray-700
              />
              <ErrorMessage name="dueDate" component="p" className="text-
            </div>

            <div className="flex items-center gap-3">
              <label className="text-gray-700 dark:text-gray-300 font-
              <Field type="checkbox" name="completed" className="w-5 h-5" />
            </div>

            <button
              type="submit"
              disabled={isSubmitting}
              className="w-full bg-green-500 hover:bg-green-600 text-white
            >
              {isSubmitting ? "Mise à jour..." : "Mettre à jour la Tâche"}
            </button>
          </Form>
        )}
      </Formik>
    </div>
  );
}
```

vous allez recréer une autre tache pour tester l'édition de la tache



ici la tache a ete modifié et marqué terminé apres validation
on est redirigé sur /dashboard



15 Conclusion

Félicitations ! Vous avez terminé la création d'une application de gestion de tâches avec Next.js. Ce tutoriel vous a montré comment construire une application moderne et réactive, en intégrant des fonctionnalités essentielles comme la gestion des tâches, la recherche, la modification, et la suppression.