**NEXTJS**

1. **Les routes**
   1. **Templates**

Dans le NextJs, il y’a un quelque chose de très intéressant, appelé **template,** Lorsque vous créez un fichier template.tsx dans un projet Next.js, ce comportement peut s'expliquer par la manière dont Next.js utilise les Layouts et les Templates. Voici pourquoi cela pourrait se produire et comment corriger ce comportement si nécessaire.

* **Comprendre les fichiers template.tsx dans Next.js**

Dans Next.js 13 et au-delà, lorsque vous utilisez l'App Router, un fichier nommé template.tsx dans un répertoire spécifique est utilisé comme un React Server Component pour fournir un "template" de rendu pour toutes les routes imbriquées dans ce répertoire.

app/

dashboard/

template.tsx

page.tsx

settings/

page.tsx

template.tsx dans dashboard/ sera appliqué pour toutes les routes sous dashboard/ (par exemple, /dashboard, /dashboard/settings).

Si vous avez un fichier template.tsx dans le répertoire racine (app/template.tsx), il sera utilisé comme template global pour toutes les routes, ce qui peut expliquer pourquoi il se "lance" dans tous les composants.

* **Solution : Limiter l’application des templates**

Si vous ne voulez pas que le fichier template.tsx soit appliqué globalement, assurez-vous de :

* + - Positionner le fichier template.tsx uniquement dans les répertoires spécifiques où il est requis.

Par exemple, si vous souhaitez qu'il s'applique uniquement aux routes sous dashboard/, placez-le dans app/dashboard/template.tsx.

* + - Supprimer le fichier template.tsx du répertoire racine (app/template.tsx) s'il existe.
* **Différence entre layout.tsx et template.tsx**
  + - **layout.tsx** : Garde l'état persistant entre les changements de pages dans son scope (comme un véritable layout).
    - **template.tsx** : Crée une nouvelle instance pour chaque navigation (utile pour éviter le partage non désiré d'état ou de composants).

La différence entre layout.tsx et template.tsx réside dans leur comportement face au rendu des enfants (ou sous-routes) et la manière dont ils gèrent l'état et la persistance.

* + - **layout.tsx** : Garde l'état persistant entre les navigations au sein de son scope.
    - **template.tsx** : Crée une nouvelle instance à chaque navigation, ce qui est utile lorsque vous ne voulez pas conserver d'état ou si vous avez besoin d’un rendu propre pour chaque navigation.

**Exemple avec une démonstration concrète**

**STRUCTURE DES FICHIERS**

app/

layout.tsx # Layout global pour toutes les routes

dashboard/

layout.tsx # Layout spécifique pour le tableau de bord

template.tsx # Template spécifique pour certaines navigations

page.tsx # Page Dashboard principale

settings/

page.tsx # Page des paramètres sous Dashboard

1. **layout.tsx - Comportement persistant**
2. export default function DashboardLayout({ children }: { children: React.ReactNode }) {
3. return (
4. <div>
5. <nav>
6. <h2>Navigation persistante</h2>
7. <ul>
8. <li><a href="/dashboard">Home</a></li>
9. <li><a href="/dashboard/settings">Settings</a></li>
10. </ul>
11. </nav>
12. <main>{children}</main>
13. </div>
14. );
15. }

Ce layout.tsx affiche une navigation persistante pour toutes les sous-routes de /dashboard (comme /dashboard et /dashboard/settings).

Lors de la navigation entre les sous-pages, le composant parent ne se démonte pas, ce qui signifie que la barre de navigation et son état restent inchangés.

1. **template.tsx - Nouveau rendu à chaque navigation**
2. // app/dashboard/layout.tsx
3. import React from "react";
4. // app/dashboard/template.tsx
5. import React from "react";
6. export default function DashboardTemplate({ children }: { children: React.ReactNode }) {
7. console.log("Template re-rendered");
8. return (
9. <div style={{ border: "1px solid blue", padding: "10px" }}>
10. <h1>Dashboard Template</h1>
11. {children}
12. </div>
13. );
14. }

* Ce template.tsx ajoute un comportement unique pour chaque navigation.
* À chaque fois que l'utilisateur navigue entre /dashboard et /dashboard/settings, le template est recréé, ce qui déclenche le re-rendu et réinitialise tout état local ou UI qui lui est associé.
* Cela est utile si vous voulez éviter que l'état persiste entre les navigations (par exemple, un formulaire).

**Quand utiliser quoi ?**

* **layout.tsx** : Pour tout ce qui doit rester constant, comme une barre de navigation, un footer, ou des éléments qui partagent un état.
* **template.tsx** : Pour des sous-routes où un nouveau rendu ou une réinitialisation complète est souhaitée. Par exemple, des formulaires où vous voulez commencer avec un état vierge.

**1.2. PATHNAME**

Nous venons d’apprendre le pathname, utilisé prinicpalement pour faire le nav-link active,

Le pathname est une partie de l'URL qui représente le chemin d'accès à une ressource sur un site web. Dans le contexte de Next.js, il est souvent utilisé pour identifier et gérer les routes ou pages sur lesquelles l'utilisateur se trouve. Cela permet de personnaliser le comportement ou l'affichage de l'application en fonction de la route active.

"use client";

import Link from "next/link";

import { usePathname } from "next/navigation";

import React from "react";

function Header() {

  const pathname = usePathname();

  return (

    <div>

      <ul id="nav">

        <li className="nav-link">

          <Link href={"/"}>

            <h1 data-name="home" className={pathname === "/" ? "active" : ""}>Home</h1>

          </Link>

        </li>

        <li className="nav-link">

          <Link href={"/dashboard"}>

            <h1 data-name="dashboard" className={pathname === "/dashboard" ? "active" : ""}>Dashboard</h1>

          </Link>

        </li>

        <li className="nav-link">

          <Link href={"/blog"}>

            <h1 data-name="blog" className={pathname === "/blog" ? "active" : ""}>Blog</h1>

          </Link>

        </li>

        <li className="nav-link">

          <Link href={"/dashboard/settings"}>

            <h1 data-name="settings" className={pathname === "/dashboard/settings" ? "active" : ""}>Settings</h1>

          </Link>

        </li>

      </ul>

    </div>

  );

}

export default Header;

**1.3. Navigation Par Ancre**

Ces liens utilisent des ancres (hash #) pour diriger l'utilisateur vers des sections spécifiques de la page, ce qui est courant dans les pages à défilement ou à contenu long.

* Alternatives pour nommer cette section :
* Navigation intra-page (ou "In-page Navigation")
* Navigation des sections
* Liens vers sections spécifiques
* Navigation ciblée par couleur (si les sections sont définies par leurs couleurs dans l'exemple).

Si cette navigation est destinée à des couleurs spécifiques, un nom plus contextuel comme "Navigation colorée" ou "Liens colorés du tableau de bord" peut être utilisé.

import Link from "next/link";

export default function Home() {

  return (

    <div>

      <h1>Home Page</h1>

      <Link href="/dashboard#yellow">Dashboard Yello part</Link>

      <Link href="/dashboard#green">Dashboard Green part</Link>

      <Link href="/dashboard#red">Dashboard Red part</Link>

    </div>

  );

}

**1.4. Use Router**

* Introduction

Le hook useRouter de Next.js fournit un accès programmatique à l'instance du routeur. Il est utilisé pour manipuler les routes dynamiquement, surveiller les changements de route et accéder à des informations telles que la route active.

* Quand l’utiliser
  + - Navigation programmatique : Quand vous devez rediriger ou naviguer vers une autre page sans utiliser un lien HTML classique (par exemple, après une action utilisateur ou en fonction d'une condition).
    - Gestion des événements : Déclencher une navigation via onClick, onSubmit, ou d'autres événements.
    - Obtenir des informations sur la route actuelle : Pour accéder à la route actuelle, des paramètres ou des requêtes.

|  |  |
| --- | --- |
| Propriété/Méthode | Description |
| router.push(url) | Navigue vers une nouvelle route (comportement équivalent à un clic sur un lien). |
| router.replace(url) | Remplace la route actuelle (sans créer une nouvelle entrée dans l'historique). |
| router.back() | Navigue vers la page précédente (équivalent au bouton "Retour" du navigateur). |
| router.reload() | Recharge la page actuelle. |
| router.query | Contient les paramètres dynamiques de la route. |
| router.pathname | Retourne le chemin de la route actuelle sans les paramètres. |
| router.asPath | Retourne le chemin complet de la route actuelle, y compris les paramètres ou le hash. |
| router.events | Permet d'écouter des événements de navigation (par exemple, début et fin d'un changement de route). |

* **A Ne pas confondre avec Link**

import Link from 'next/link';

function HomePage() {

  return (

    <div>

      <Link href="/about">

        <a>Go to About Page</a>

      </Link>

    </div>

  );

}

export default HomePage;

|  |  |  |
| --- | --- | --- |
| Caractéristique | useRouter | Link |
| Type | Hook React | Composant Next.js |
| Navigation programmatique | Oui (navigation déclenchée par du code) | Non |
| Navigation déclarative | Non | Oui (lien HTML classique) |
| Requiert un événement utilisateur | Non (peut être utilisé sans interaction directe) | Oui (nécessite un clic pour naviguer) |
| Optimisé pour SPA | Oui | Oui |
| Simplicité d'utilisation | Moins intuitif (nécessite du code supplémentaire) | Plus simple pour des liens statiques |

* **Quand choisir useRouter ou Link**

**Utilisez useRouter si :**

* Vous devez naviguer en fonction d'une logique conditionnelle ou d'un événement utilisateur.
* Vous souhaitez accéder à des informations sur la route actuelle ou les paramètres de requête.
* Vous gérez des événements complexes comme la redirection après une action.

**Utilisez Link si :**

* Vous voulez des liens simples pour naviguer entre des pages.
* Vous construisez des menus ou une navigation classique.

import Link from 'next/link';

import { useRouter } from 'next/router';

function Header() {

  const router = useRouter();

  const handleSpecialNavigation = () => {

    if (router.pathname === '/home') {

      router.push('/special');

    } else {

      router.push('/default');

    }

  };

  return (

    <div>

      <nav>

        <Link href="/home">

          <a>Home</a>

        </Link>

        <Link href="/about">

          <a>About</a>

        </Link>

        <button onClick={handleSpecialNavigation}>Special Navigation</button>

      </nav>

    </div>

  );

}

export default Header;

**1.5 useSearchParams**

* Un hook de Next.js pour récupérer et interagir avec les paramètres de recherche dans l'URL (la partie après le ?).
* Accessible uniquement dans les composants côté client (d'où la directive "use client";).

"use client";

import { useSearchParams } from "next/navigation";

export default function Home() {

  const searchParams = useSearchParams();

  console.log(searchParams.get("search"))

  return (

    <div>

      <h1>Home Page</h1>

    </div>

  );

}

Dans la console, j’aurais comme output : bike

**1.6. Dynamic Routes**

Imaginons que nous ayosn envie de faire une requete de type SingleId

// app/product/[productId]/page.tsx

export default async function ProductPage({ params }: { params: { productId: string } }) {

  const { productId } = params

  const product=await fetch(`https://dummyjson.com/products/${productId}`).then(res=>res.json())

  console.log(product)

  // Tu peux utiliser productId pour afficher la page

  return <div>{product.title}

  <h3>{product.description}</h3>

  <p>{product.price}</p>

  </div>;

}

**THEORIE DU STREAMING**

Le streaming en Next.js est une fonctionnalité puissante introduite avec le App Router (Next.js 13 et versions ultérieures), qui permet de rendre les pages de manière progressive. Cela améliore les performances et l'expérience utilisateur en affichant des parties d'une page dès qu'elles sont prêtes, plutôt que d'attendre que tout le contenu soit généré avant le rendu.

**Concepts clés du streaming en Next.js**

1. ***Rendu Server-Side Streaming (SSR Streaming) :***
   * + Avec le streaming, le contenu est généré par le serveur et envoyé au client en plusieurs morceaux.
     + Dès qu'une partie est prête, elle est rendue côté client, même si d'autres parties de la page sont encore en cours de traitement.
     + Cela réduit le temps nécessaire pour afficher les parties critiques d'une page, améliorant ainsi le Time to First Byte (TTFB) et le Largest Contentful Paint (LCP).
2. ***Suspense pour le streaming :***
   * + React introduit le composant Suspense qui permet de différer le rendu d'une partie de l'interface utilisateur jusqu'à ce que les données nécessaires soient disponibles.
     + Avec Next.js, Suspense est intégré au streaming pour afficher des fallbacks pendant le chargement de certaines parties de la page.

**NEXT/FONT : Une Gestion Optimisée des Polices dans Next.js**