ATELIER 5 : STREAMING ET SUSPENSE NEXT JS

Introduction

Avec SSR (Server Side Rendering), il y a une série d'étapes qui doivent être complétées avant qu'un utilisateur puisse voir et interagir avec une page :

1. Tout d'abord, toutes les données d'une page donnée sont récupérées sur le serveur.
2. Le serveur restitue ensuite le code HTML de la page.
3. Les codes HTML, CSS et JavaScript de la page sont envoyés au client.
4. Une interface utilisateur non interactive est affichée à l'aide du code HTML et CSS généré.
5. Enfin, React hydrate l'interface utilisateur pour la rendre interactive.

L'hydratation est le processus d'utilisation de JavaScript côté client pour ajouter l'état de l'application et l'interactivité au code HTML rendu par le serveur.

React s'attachera au HTML qui existe à l'intérieur du domNode et prendra en charge la gestion du DOM à l'intérieur. Une application entièrement construite avec React n'aura généralement qu'un seul appel d'hydrate avec son composant racine.

Ces étapes sont séquentielles et bloquantes, ce qui signifie que le serveur ne peut restituer le code HTML d'une page qu'une fois que toutes les données ont été récupérées. Et, sur le client, React ne peut hydrater l'interface utilisateur qu'une fois le code de tous les composants de la page a été téléchargé.

SSR avec React et Next.js aide à améliorer les performances de chargement perçues en montrant une page non interactive à l'utilisateur dès que possible.

Cependant, cela peut encore être lent car toutes les récupérations de données sur le serveur doivent être terminées avant que la page puisse être affichée à l'utilisateur.

Streaming s'agit, lorsque nous avons une page et qu'on fait le load à partir de ressources différentes, comme par exemples les produits et les commentaires. Donc on aura un loading différent pour les produits et un loading différent pour les commentaires. Chaque loading occupe une partie de la page et non la page entière.

Le streaming vous permet de décomposer le code HTML de la page en plus petits morceaux et d'envoyer progressivement ces morceaux du serveur au client.

Cela permet d'afficher des parties de la page plus tôt, sans attendre que toutes les données soient chargées avant qu'une interface utilisateur puisse être rendue.

Le streaming fonctionne bien avec le modèle de composants de React car chaque composant peut être considéré comme un morceau.

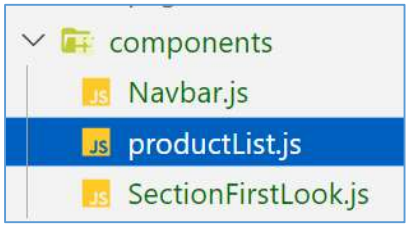
Pour un contrôle plus granulaire, vous pouvez envelopper vos propres composants dans une limite React **Suspense**. <Suspense> fonctionne en enveloppant un composant qui effectue une action asynchrone (par exemple, récupérer des données), en affichant une interface utilisateur de secours (par exemple, template, spinner, etc.) pendant qu'elle se produit, puis en échangeant votre composant une fois l'action terminée.

En utilisant Suspense, vous bénéficiez de :

- Streaming Server Rendering - Rendu progressif du HTML du serveur au client.
- Hydratation sélective - React donne la priorité aux composants à rendre interactifs en premier en fonction de l'interaction de l'utilisateur.

Appel du composant productList

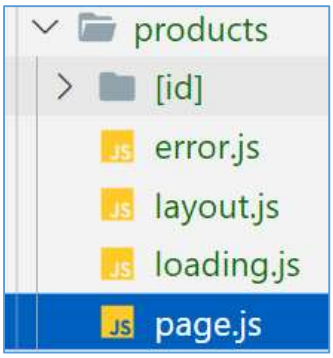
Créer le fichier productList.js sous le répertoire components. On va y mettre le code du return de product/page.js




```
import React from 'react';
import Link from 'next/link';


const ProductList= async ({products})=> {

    return (
        <section style={{ backgroundColor: "#eee" }}>
        <div className="container py-5">
          <div className="row">
            {products?.map((product) => (
              <div
                key={product?.id}
                className="col-md-12 col-lg-3 mb-4 mb-lg-0 pt-4 "
              >
                <div className="card h-100">
                  <img
                    src={product?.images[0]}
                    className="card-img-top p-5"
                    height={320}
                    width={100}
                    alt={product.title}

                  />
                  <div className="card-body">
                    <div className="d-flex justify-content-between">
                      <p className="small">{product?.category.name}</p>
                    </div>

                    <div className="d-flex justify-content-between mb-3">
                      <Link href={`/products/${product?.id}`}>
                        <h5 className="mb-0">{product?.title}</h5>
```

```
                    </Link>
                    <h5 className="text-dark mb-0">${product?.price}</h5>
                  </div>
                </div>
              </div>
            </div>
          ))}
        </div>
      </div>
    </section>
  );
}

export default ProductList;
```

Modifier products/page.js qui fera appel à components/productList.js



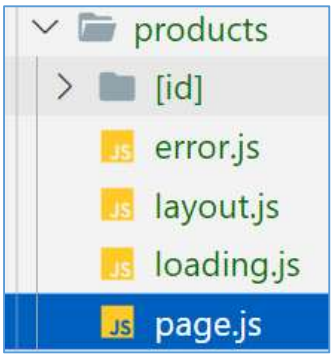

```
import React from 'react';
import ProductList from '@/components/productList';

async function getProducts(){
    const res= await fetch('https://api.escuelajs.co/api/v1/products')
    const products = await res.json();
    return products;
}

const ProductsPage= async ()=> {
    const products = await getProducts();

    return (
      <ProductList products={products} />
    )
}

export default ProductsPage;
```

Appel du composant categoryList

Créer le fichier components/categoryList.js


```
import React from 'react';

const CategoryList= async ({categories})=> {

    return (
  <section style={{ backgroundColor: "#eee" }}>

            {categories?.map((category) => (
              <div key={category?.id}>
              {category?.name}
              </div>
            ))}

      </section>
     );
}

export default CategoryList;
```

Modifier products/page.js en faisant appel à CategoyList.

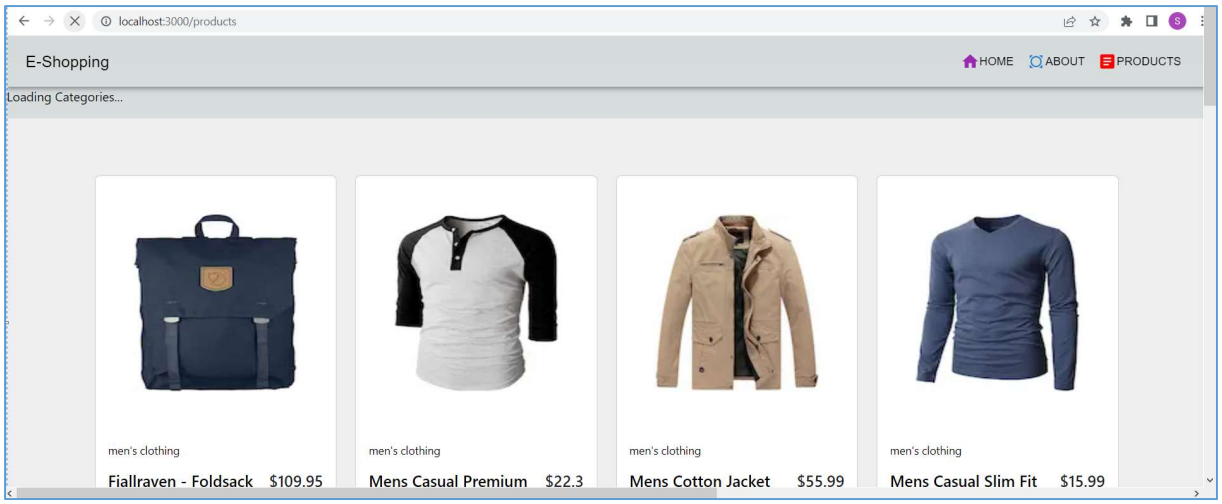


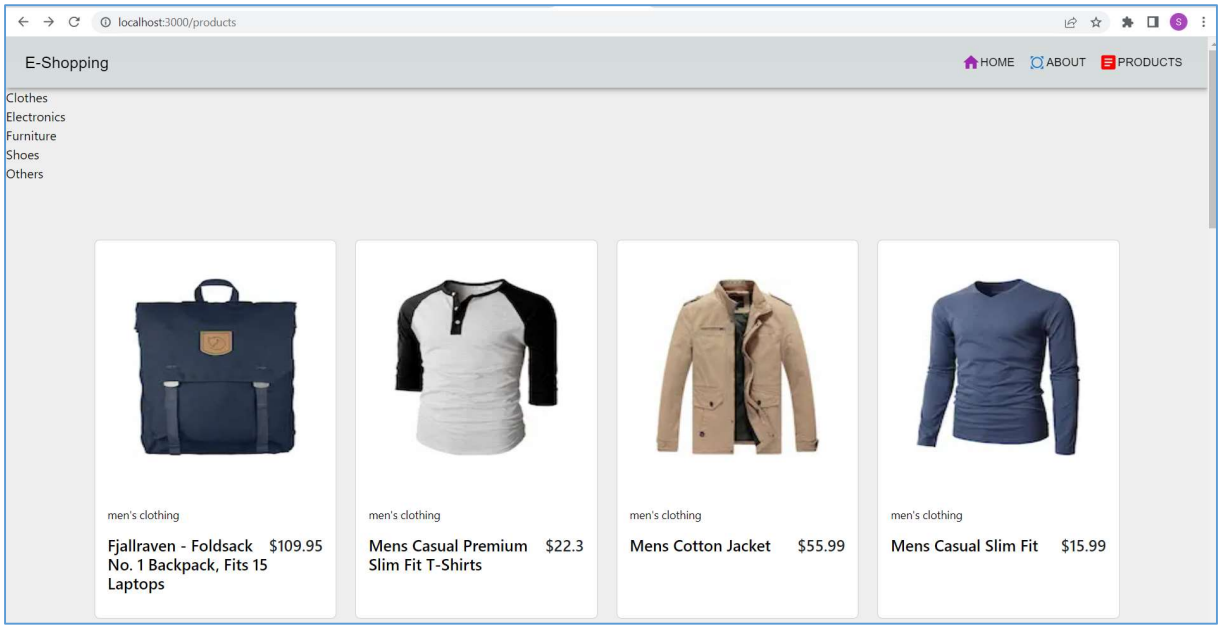
Nous pouvons gagner du temps en lançant les deux requêtes en parallèle, cependant, l'utilisateur ne verra pas le résultat rendu tant que les deux promesses ne seront pas résolues.

On va rendre le streaming plus user-friendly (convivial), on va ajouter **Suspense**. On utilise Suspense pour améliorer l'expérience utilisateur.

Suspense peut être utilisé pour fournir une interface utilisateur de secours pour les composants asynchrones qui ne sont pas encore résolus, afin qu'ils puissent s'afficher sans bloquer. Suspense permet de décomposer le travail de rendu et afficher une partie du résultat dès que possible.

```jsx
import React, { Suspense } from "react";
import ProductList from '@/components/productList';
import CategoryList from '@/components/categoryList';

async function getProducts(){
    const res= await fetch('https://api.escuelajs.co/api/v1/products')
    const products = await res.json();
    return products;
}

async function getCategries(){
  const res= await fetch('https://api.escuelajs.co/api/v1/categories')
  const categories = await res.json();
  return categories;
}

const ProductsPage= async ()=> {
    const products = await getProducts();

    const categories = await getCategries();

    return (
      <>

      <Suspense fallback={<p>Loading Categories...</p>}>
      <CategoryList categories={categories} />
      </Suspense>

      <Suspense fallback={<p>Loading Products...</p>}>
      <ProductList products={products} />
      </Suspense>

    </>

    )
}

export default ProductsPage;
```





Affichage des catégories avec le Carousel

Installer la dépendance :

```
npm i react-responsive-carousel --legacy-peer-deps
```

Référence : https://www.npmjs.com/package/react-responsive-carousel

Modifier le code de categoryList.js

Remarque :

Le composant ne doit pas être déclaré asynchrone.

```
"use client" ;
import React from 'react';
import "react-responsive-carousel/lib/styles/carousel.min.css";
import { Carousel } from 'react-responsive-carousel';

const CategoryList=  ({categories})=> {

    return (
    <center>
      <Carousel width="30%">
         {categories?.map((category) => (
              <div key={category?.id}>
               <img src={category?.image} alt="image"/>
                      <p className="legend">{category?.name}</p>
              </div>
            ))}
      </Carousel>
   </center>

     );
}

export default CategoryList;
```